Automated Testing: Requirements Propagation via Model Transformation in Embedded Software

Nader Kesserwan

A Thesis
In
The Concordia Institute
For
Information System Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Information and Systems Engineering) at
Concordia University
Montreal, Quebec, Canada

March 2020

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Nader Kesserwan

Entitled: Automated Testing: Requirements Propagation via Model Transformation in Embedded Software

and submitted in partial fulfillment of the requirements for the degree of

Doctor Of Philosophy (Information & Systems Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Mehdi Hojjati

_____ External Examiner
Dr. Esma Aïmeur

_____ External to Program
Dr. Ferhat Khendek

_____ Examiner
Dr. Roch Glitho

_____ Examiner
Dr. Abdessamad Ben Hamza

_____ Thesis Co-Supervisor
Dr. Rachida Dssouli

_____ Thesis Co-Supervisor
Dr. Jamal Bentahar

Approved by _____
Dr. Mohammad Mannan, Graduate Program Director

March 3, 2020 _____
Dr. Amir Asif, Dean
Gina Cody School of Engineering & Computer Science

ABSTRACT

Automated Testing: Requirements Propagation via Model Transformation in Embedded Software

Nader Kesserwan, Ph.D.

Concordia University, 2020

Testing is the most common activity to validate software systems and plays a key role in the software development process. In general, the software testing phase takes around 40-70% of the effort, time and cost. This area has been well researched over a long period of time. Unfortunately, while many researchers have found methods of reducing time and cost during the testing process, there are still a number of important related issues such as generating test cases from UCM scenarios and validate them need to be researched.

As a result, ensuring that an embedded software behaves correctly is non-trivial, especially when testing with limited resources and seeking compliance with safety-critical software standard. It thus becomes imperative to adopt an approach or methodology based on tools and best engineering practices to improve the testing process. This research addresses the problem of testing embedded software with limited resources by the following.

First, a reverse-engineering technique is exercised on legacy software tests aims to discover feasible transformation from test layer to test requirement layer. The feasibility of transforming the legacy test cases into an abstract model is shown, along with a forward engineering process to regenerate the test cases in selected test language.

Second, a new model-driven testing technique based on different granularity level (MDTGL) to generate test cases is introduced. The new approach uses models in order to manage the

complexity of the system under test (SUT). Automatic model transformation is applied to automate test case development which is a tedious, error-prone, and recurrent software development task.

Third, the model transformations that automated the development of test cases in the MDTGL methodology are validated in comparison with industrial testing process using embedded software specification. To enable the validation, a set of timed and functional requirement is introduced. Two case studies are run on an embedded system to generate test cases. The effectiveness of two testing approaches are determined and contrasted according to the generation of test cases and the correctness of the generated workflow. Compared to several techniques, our new approach generated useful and effective test cases with much less resources in terms of time and labor work.

Finally, to enhance the applicability of MDTGL, the methodology is extended with the creation of a trace model that records traceability links among generated testing artifacts. The traceability links, often mandated by software development standards, enable the support for visualizing traceability, model-based coverage analysis and result evaluation.

Acknowledgments

Foremost, I would like to express my sincere gratitude to my supervisor Dr. Rachida Dssouli for the continuous support, patience, motivation and advice. Her guidance helped me conducting this study, communicating with academics and writing of the Thesis. I could not have imagined having a better or friendlier supervisor.

It is also an honor for me to express my deep thanks to my co-supervisor Dr. Jamal Bentahar. His valuable and immense comments and advices gave me more insight through the research area.

I would also like to thank my committee members, Dr. Ferhat Khendek, Dr. Roch Glitho, Dr. Abdessamad Ben Hamza for serving as my committee members. I also want to thank you for letting my defence be an enjoyable moment, and for your brilliant comments and suggestions, thanks to you.

I am also grateful to Dr. Bernard Stepien in University of Ottawa, his help in offering me technical advices in building tools is much counted and appreciated.

I wish to extend my warmest thanks to Pierre Labrèche manager at Esterline CMC Electronics who have helped me with my research in offering me advices, lab access and providing me with all needed documents. Wishing you all the best in your new retired life.

I owe my loving thanks to my wife Rima and my kids Rami, Sami and Nada. It would have been impossible for me to finish this study without their continuous support, encouragement and understanding.

Table of Content

List of Tables

List of Figures

List of Illustrations

Abbreviation

ATC	Abstract Test Case
ATS	Abstract Test Suites
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
ETC	Executable Test Case
ES	Embedded System
FMS	Flight Management System
HLR	High Level Requirement
LGS	Landing Gear System
LHS	Left-Hand Side
LLR	Low Level Requirement
MBT	Model-Based Testing

MDD	Model-Driven Development
MDT	Model-Driven Testing
MDTGL	Model-Driven Testing on different Granularity Level
NL	Natural Language
RHS	Right Hand Side
RTCA	Radio Technical Commission for Aeronautics
SUT	System Under Test
SVP	Software Verification Process
TC	Test Case
TDL	Test Description Language
TTCN-3	Testing and Test Control Notation
UCM	Use Case Map

Chapter 1 Introduction

1.1 Research Motivation

As software systems become increasingly complex, the demand for software verification grows. Testing is a major cost factor during software development, sometimes consuming more than 50% of the overall development effort [1], [2]. To address growing demand, many testing approaches and strategies have been developed with the aim of minimizing cost and achieving high fault detection capabilities. One of the most promising approaches is model-based testing (MBT). This approach can reduce test costs due to its ability to capture and validate system behaviour from an early stage of the software development cycle; it also promotes the use of tools to automate the process of test case generation, execution, and evaluation [3]. The process of MBT relies on building models to represent system requirements. These models, therefore, form an efficient source for deriving test cases. According to a 2011 survey in the car industry [4], "Model-based testing (i.e. the generation of test cases out of a test model) is currently not used intensively. Only 35% of the participants use it right now, but almost 50% plan to use it in the near future".

Another promising technique is model-driven testing [5] (MDT), which is an automation of MBT that uses model-transformation technology on formal models, their meta-models, and transformation rules defined in terms of mappings between the elements of meta-models. Automatic model transformations play a critical role in model-driven engineering (MDE) since they automate complex, tedious, error-prone, and recurrent software development tasks [6], [7], [8]. The key challenge of MDT is to transform higher-level models to platform-specific models that tools can use to generate code. Examples of transformations are a refinement of a design model by adding details pertaining to a particular target platform, refactoring a model by changing its structure to enhance design quality, or reverse engineering code to obtain an abstract model.

A good candidate of a higher-level model is a one expressed in the modeling notation called Use Case Maps (UCM). This modeling language uses paths that causally link activities (called responsibilities), which can be bound to underlying organizational structures [9], [10]. The UCM

scenario meta-model can be used to model service requirements and high-level designs for reactive and embedded systems (ESs). It is, therefore, a natural candidate for use in the process of generating requirements-directed test suites. Goal models capture hierarchical representations of stakeholder objectives, requirements, possible solutions, and their relationships to help requirements engineers understand stakeholder goals and explore solutions based on their impact on these goals [11]. Although, several approaches have been suggested to improve UCM-based testing by deriving test goals [12], [13], [14], [15], [16] its abstraction level remains inappropriate for the generation of implementation-level test cases. The UCM models emphasize behavior rather than data, and also abstract from detailed communication mechanisms which make deriving executable test cases (ETC) a difficult activity. The abstraction gap that resides between the simple expression of a UCM test purpose and the complex coding of executable test scripts needs to be filled by an intermediate representation that can be the starting point for test automation. In [17], The traversal mechanism prototyped in jUCMNav's tool [18] is used to transform the test purposes into test specification packages represented as XML elements. The exported representation did not handle the combinations of scenarios or alternative behavior nor has been validated or transformed into scripting language. Our approach and its supporting techniques have been validated against an industrial embedded system. The absence of an alternative element in the UCM scenario metamodel has been resolved.

Another challenge besides transforming UCM scenario models to test cases in a scripting language is the validation of the transformation, both in terms of technical correctness and usefulness. The test case generation task is critical and thus the model transformations that automate it must be validated. A fault in a transformation can introduce a fault in the transformed model, which if undetected and not removed, can propagate to other models in successive development steps. As a fault propagates across transformations, it becomes more difficult to detect and isolate. Since model transformations are meant to be reused, faults present in them may result in many faulty models.

The variety of different models produced in the transformation process discussed in the previous section poses challenges to requirements traceability and assessment. This diversity of artifacts results in an intricate relationship between requirements and the various models. The role played by relationships among artifacts to support automation of testing activities had long been

recognized; relationships from behavioral models to test cases and from test cases to test results support coverage measurement, result evaluation, and selective regression testing. The creation and maintenance of explicit relationships among test-related artifacts is, therefore, the main challenge to the automated support of such activities. Over the past years, traceability—the ability to describe and follow the life of software artifacts [19]— has gained in importance and used as a quality attribute for software. Requirements traceability is often mandated by software development standards. It is required to support activities such as result evaluation, regression testing, and coverage analysis. In addition to test generation, challenges to MBT include creation and maintenance of traceability information among test-related artifacts, time challenge and system safety that is set very high by regulatory authorities such as radio technical commission for Aeronautics (RTCA) [20].

As a result, there is an obvious need to generate executable test cases from UCM scenarios and to validate the model transformation from requirement level to implementation level via an intermediate level that bridges the gap between the two levels. Further research is also needed to link the intricate relationships among test-related artifacts, obtained as a product of the transformation, to support the automation of testing activities such as coverage measurement and result evaluation.

In this context, the following issues should be addressed:

- Construction of a test development process composed of three phases where each phase represents a different level of test abstraction expressed by an appropriate language.
  - UCM notation to model the complexity of the SUT (test purposes) used as a base to derive test specifications.
  - Test definition notation to specify test description (test specifications) such as the test description language (TDL) that can be used as a base to derive test cases.
  - Scripting language to implement and execute a test case (test implementation)
- Development of model-driven testing methodology that generates test cases through a model transformation based on the selected languages
  - Determine and resolve the divergence that obstacles the transformations between the three languages. These obstacles can be related to a lack of suitable abstraction for specifying transformations. Consequently, transformations can be hard to write,

comprehend, and maintain. For instance, develop a data model to address the lack of data in the UCM scenario that is needed in a test case and resolve the differences between UCM, TDL, and TTCN-3 (test configuration and alternative behavior). In addition, performing a model transformation requires a clear understanding of the abstract syntax and semantics of both the source and target.

  - Demonstrate the feasibility of the transformation using industrial software tests.
- Automation of the model transformation and prototyping it into tools.
- Maintaining traceability links among generated test artifacts by developing a traceability framework that automatically links the intricate relationships among test-related artifacts.
- Applying and validating the model transformation that generates the test cases to industrial ESs both in terms of technical correctness and usefulness.

The aforementioned themes; (1) generate test cases in TTCN-3 from UCM models using model transformation, (2) validate the model transformation in the avionic industry, and (3) maintain traceability links among test-related artifacts play an important role in the thesis chapters and contents. The next section discusses the new approach and the objectives for conducting this study which leads to the set of stated contributions (Section 1.3).

1.2 New Approach: MDTGL

In this thesis, we present an innovative approach where we generate test cases in a language called testing and test control notation (TTCN-3) [21] from test specifications described by TDL [22]. The TDL test specifications in their turn are generated from test purposes enclosed in a semiformal visual notation for causal scenarios called UCMs.

TTCN-3 is a test specification language designed for specifying test cases to be implemented and executed against SUT. TTCN-3 is selected for its industrial strength and for its applicability to a variety of application domains and levels of testing.

TDL can be used as an intermediate representation to describe scenarios on a lesser abstraction level than high-level test description but on a higher abstraction level than scripting languages. We believe that using TDL in a scenario-oriented approach help close the abstraction gap that resides between test purposes and test cases.

We believe that using UCMs in a scenario-oriented approach represents a judicious choice for the description of communicating and ESs. They fit well in the design approach proposed in this thesis, the MDTGL methodology.

Considering the research motivation discussed in Section 1.1, the aim of this thesis is to provide techniques to generate test cases in a better way where resources are limited, through model transformation and refinement. It also intends to validate the generated artifacts in terms of usefulness and effectiveness and create traceability links among the generated artifacts. It should fill the gap between the stage where functional requirements are described abstractly and their implementation details handled by test cases.

To fulfill this aim, a number of objectives are necessary:

Objective 1: To determine the differences and obstacles that reside among the three languages; UCM, TDL, and TTCN-3.

Objective 2: To resolve the obstacles and differences that exist among the three languages and demonstrate the approach feasibility.

Objective 3: To generate test cases in TTCN-3 from UCM models via TDL based on requirement analysis, model transformation, and refinement process.

Objective 4: To align the traceability requirement with generated test artifacts and testing.

Objective 5: To validate the generated testing artifacts in terms of effectiveness and usefulness at the specification and implementation level.

Objective 6: To develop and provide traceability evidence from requirements to tests for compliance with DO-178C standards.

The thesis presents a methodology where the transformation of requirements to test cases is different from the one used by the most popular techniques. The approach focuses on transforming the highly abstract test goals into concrete test cases. A prime goal of this thesis is hence to enable the generation of test cases and validate them in terms of correctness, usefulness, and effectiveness.

MDTGL aims to improve the maturity of test case generation processes based on model transformation by introducing a model transformation technique among three languages representing tests from high-level abstraction to low-level scripting language. **Figure** 1.1 presents such an approach and introduces the main concepts behind the MDTGL. The key points of the MDTGL methodology are: (1) natural language (NL) requirements are described in UCM behavioral models; (2) These models are transformed to test goals, and then based on developed rules, to abstract test cases (ATC) in TDL notation that are completed manually with test objectives and data instances; and (3) the obtained ATCs are transformed, based on developed rules, along with concrete test data to test cases (TC) in TTCN-3 language.

The approach can be seen as a process of successive refinements of specifications that involves model transformation and the insertion of additional information.

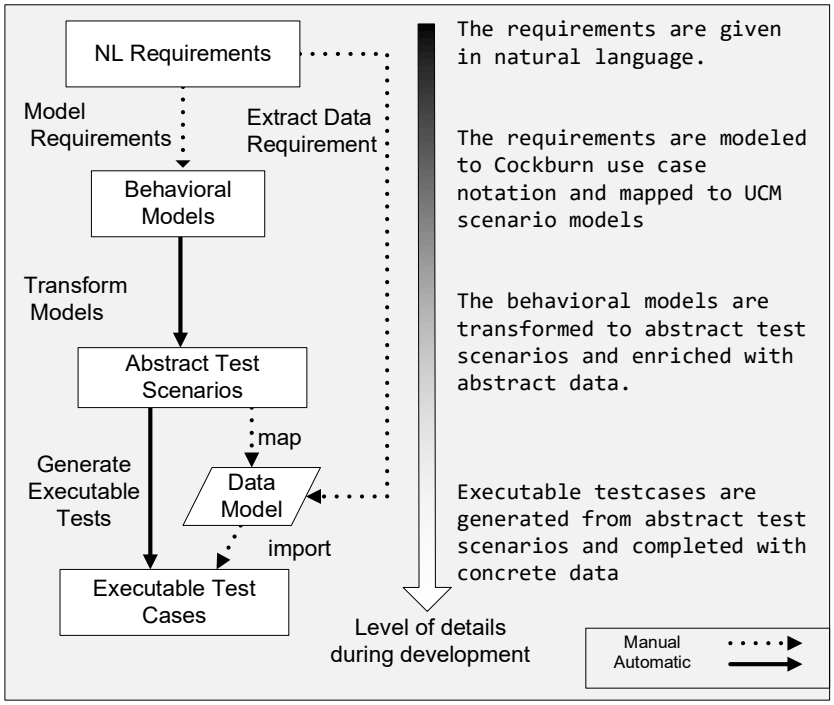


Figure 1.1: MDTGL methodology

1.3 Thesis Contribution

This thesis offers four main contributions: (1) the reverse engineering work to help build the *MTDGL* methodology from legacy software tests, (2) the development and the extension of the

MTDGL methodology, (3) a set of techniques to support the *MTDGL* cycles, and (4) the application of *MTDGL* to validate the generation of testing artifacts.

1.3.1 Contribution 1: Reverse-Engineering the Legacy Software Tests to Model-Driven Testing

In order to support test automation and to reduce the effort involved in testing, our starting point was to restructure legacy software tests developed manually to be driven from models. Our reverse-engineering process achieved the following goals:

Help build the model-driven testing methodology: we automatically structured legacy software tests to a model-driven testing methodology, based on formalized test cases. The legacy test cases are initially translated to TTCN-3 code and then abstracted to TDL models. The goal here is to study model-driven test case generation from TDL and to evaluate TDL as a formal language for expressing test cases. Reaching this point, the feasibility of transforming TTCN-3 scripts into a TDL model is shown, and a forward engineering process to regenerate the test cases can be undertaken.

1.3.2 Contribution 2: *MTDGL* Methodology

We claim that *MTDGL* methodology has several benefits, difficult to find all at once in other design and standardization processes:

- **Reducing Test Effort and Start Testing Early**: since software requirements are described in UCM scenarios and transformed to test cases, the test development phase is minimized. The TCs are no longer written by hand or manually corrected, but generated using model transformation which reduces the number of iterations to get them correct. We validated the MTDGL methodology against an industrial embedded system. Furthermore, the test engineers don't need to wait; they describe the requirements in the scenario model and then push a button to generate the tests.
- **Test Case Generation**: scenarios guide the generation of test cases, hence allowing the verification of the prototype against the UCMs and its validation against the informal functional requirements. The test suite can itself be validated using structural coverage criteria on the model. It can be reused as a basis for functional or regression test suite in

the subsequent steps of the development process. The validation of the generated test cases is covered in Chapter 6.

- **Requirement Traceability**: documentation can be generated from the model and is thus consistent with the tests. Since TCs are derived from the UCM models where requirements are described, any defect found during the execution of a TC can be traced back to its requirement. The section Traceability Links Framework in Chapter 5 extends the MDTGL methodology to create explicit relationships in a trace model among testing artifacts.
- **Systematic**: with the help of the developed tools, repeated tests are enabled which ensures the robustness of the test results. The result obtained from the Experimental Method section in Chapter 6 demonstrates the robustness of the test results.
- **Design Documentation and System Understandability**: the documentation of requirements and designs is done as we go along the development cycle. The generated test specification in TDL can be used mainly for communication between stakeholders as the basis for implementing concrete tests. It should also be understandable by non technical people who do not have to know every technical detail described in the test specifications. UCMs allow different specialists to become involved in discussions at different levels while sharing a common language and, hopefully, understanding.

1.3.3 Contribution 3: Theories and Techniques Supporting

Different theories and techniques are involved in the support of the MDTGL cycles. The developed techniques in this thesis are:

Construction of TDL Specifications from UCMs: in his work, Boulet provided a mapping between UCM paths and TDL packages expressed as XML elements. This mapping is extended in this thesis to build a valid test specification based on TDL metamodel, which better reflect the test semantic, and to be used as a base to derive test cases.

Automated the Absence of Alternative Behavior: In our approach, we resolved the absence of alternative elements in the UCM scenario metamodel. The metamodel of the UCM exported early in the process doesn't have an alternative element that normally a test case has to handle alternate test behavior. We developed a technique to automate the post-processing of the interaction

behavior. Our automated tool selects the common interaction behavior that represents different responses to the tester and groups them in the alternative element.

Automated Development of Testing Artifacts: the thesis presents a new technique for automatically generate test cases using model transformation between UCM, TDL, and TTCN-3. The differences and the abstraction that exits among the three languages are resolved and automated via transformation and refinement process.

1.3.4 Contribution 4: Illustrative Experiments Validating *MTDGL*

The *MDTGL* approach and its supporting techniques have been validated against an industrial ES. Chapter 5 includes results and lessons learned from real case study experiment:

- **Technical Feasibility**: the technical feasibility of the *MTDGL* is demonstrated via a case study from the avionics public domain for the generation of TCs from TDL specifications.
- **Test Suite Validation**: The evaluation of the MDTGL methodology is sampled with an industrial product from the private domain to validate the various test suites generated using the UCM scenarios. These experiments discuss the efficiency and performance of the test suites in comparison with the industrial testing approach according to three assessment criteria (requirement coverage, the correctness of generated workflow and its cost). Most of these experiments were done in collaboration with industrial partners, professors, and engineers. The MDTGL approach has been evaluated by our research partner to replace its current testing process.

1.3.5 Issues Not Addressed in this Thesis

There are a couple of important issues that the MDTGL methodology do not address in this thesis:

- The automated generation of test input needed in test cases from UCMs is not a goal of this thesis.
- The testing used here is functional (black-box). It is targeted towards the user-system level. Component or unit testing is not addressed in the thesis.

1.4 Thesis Outline

The rest of the Thesis is structured as follows.

Chapter 2 emphasizes the importance of testing ESs behaviour. The chapter defines testing and presents an overview of testing types according to the three-dimension model. Manual testing suffers from a high cost in terms of time, effort and resources due to the growing complexity of ESs. This suggests the potential benefits of applying modeling and model transformation in a testing context. MBT and MDT can thus be used to describe software specification in a behavioral model to automatically derive test cases. The latter are completed when necessary to be executed on the SUT with the aim to find any potential misbehavior.

As an important activity to cut down the cost of manual testing, this chapter discusses test case generation techniques such as model-based, specification-based and natural language techniques. The chapter discusses the use of traceability in the context of requirements engineering and model-drive. Next, it highlights the importance of aligning the requirement traceability with testing. A set of related work for each testing activity; (1) model transformation, (2) test case generation, and (3) requirement traceability are presented and discussed to highlight the research motivation of this thesis.

Chapter 3 introduces a background chapter where the three domain-specific languages used in the model transformation approach are introduced. The construct of each language is described extensively with examples.

Chapter 4 in this chapter, a reverse engineering process to help build the new testing methodology is presented. The reverse engineering process started with a migration of legacy test cases, written as Ant/XML files, into the TTCN-3 code and are reengineered with data to a higher level of abstraction to obtain abstract test cases in TDL notation. Our overarching goal is to support test automation and discover a path from TDL to TTCN-3.

Chapter 5 proposes a novel testing methodology to support the testing of ES by generating test cases from a description of the abstract tests (derived from behavioral models), and maintaining requirement traceability. The methodology called MDTGL and it is based on requirement analysis and model transformation where the main goal is to automate the generation of test

artifacts. The new technique develops and validates tools for automating the generation of test cases based on model transformation.

The chapter presents and discusses the new approach in great details, it also demonstrates its feasibility by applying it to an avionic public case study.

Chapter 6 The evaluation of the MDTGL methodology is sampled with an industrial product from the private domain to validate its efficiency and performance in comparison with the industrial testing approach according to three assessment criteria (requirement coverage, the correctness of generated workflow and its cost). As a result, the chapter presents an experiment applied to the avionics case study for estimating the assessment criterion. A discussion with generalization of the approach and set of lessons learned showing the difficulties encountered especially for testing ES is then highlighted.

Chapter 7 summarizes the research contributions and findings. Finally, the chapter describes the limitations of this study and opportunities for future work.

Chapter 2 Literature Review

2.1 Topic Overview

The role of computing devices, embedded in everyday objects, has grown tremendously over the last two decades. Our modern society is hugely dependent on ESs to monitor or control different hardware infrastructures [23]. To give an example, a typical car produced at the beginning of the 1990-ies was largely a mechanical unit. Today, a large part of the development costs in a typical front-edge car manufacturing company are related to software development. 'Embedded system' is a generic term that refers to computerized systems interacting closely with the real world through sensors, networks and actuators [24], [25]. Systems like mobile phones, flight management systems, air traffic control systems, patient monitoring systems, and many others can be considered as examples of ESs [26].

Software is one of the cores and most error-prone components of ESs. Any failures encountered can range from a slight system aberration (e.g., coffee machine malfunction) to financial loss and even loss of human life (e.g., in safety-critical systems) due to misbehavior. Thoroughly checking the correctness of ES's software before deployment using various validation activities (e.g., testing) therefore becomes necessary [27].

The rest of the chapter is organized as follows. Section 2.2 introduces the concept of software testing. Section 2.3 highlights some of the testing categories according to the three-dimension model. Section 2.4 presents the principles of MDA whereas Section 2.5 presents the principles of MBT. Section 2.6 gives some definitions, presents model transformation categories along with design features and surveys work done on the UCM model transformation. Section 2.7 presents the various test case generation techniques that were used in the literature as a mechanism to automate the development of tests to overcome some testing problems such as high cost and labor-intensive. Section 2.8 discusses requirement traceability and its important role in coverage analysis and result evaluation. Section 2.9 concludes the chapter.

2.2 Software Testing

Testing is a systematic process of finding software errors by running the software in a controlled environment and analyzing its outcomes before its deployment. The process of software testing involves the generation and execution of test cases on software [28]. The generated test cases need to be executed on the SUT to collect the produced outputs. The observed outputs are then analyzed and compared with those expected according to a derived test oracle. A test oracle can be defined as the rules by which the expected and actual outputs are compared to decide whether the SUT is correct or not [29].

One strategy which significantly reduces the test cost is to decrease human involvement and automate the test process through the use of verified testing tools [30] To address growing demand, several new technologies have emerged to help with the development and verification of high-quality systems.

2.3 Testing Types

Moreover, different test types can concentrate on various SUT aspects and can be performed at several levels to increase the overall confidence about its quality [31]. **Figure 2.1** depicts different types of testing categorized in three dimensions (i.e., testing level, testing accessibility and testing aspects). Note that different types of testing can be performed together[32].

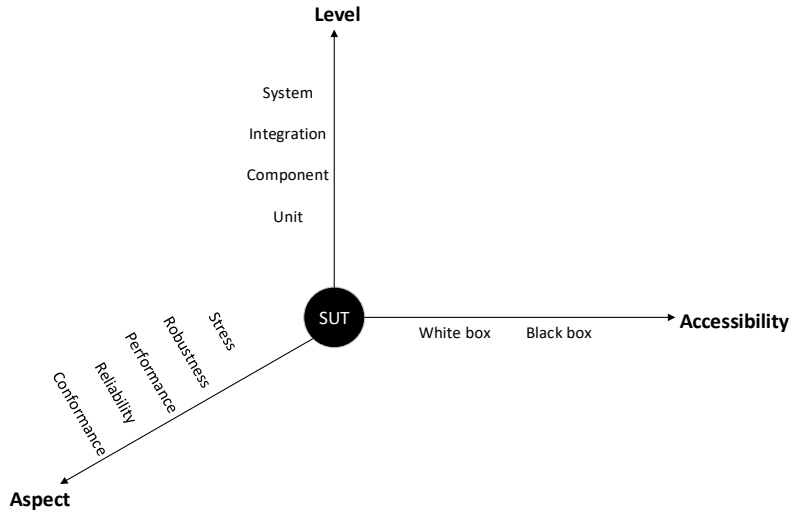


Figure 2.1: Testing types (Briones, 2007)

With respect to which level of the SUT testing is applied, four types of testing can be identified: unit, component, integration, and system-based testing. Unit testing checks the correctness of the smallest unit of the SUT alone (e.g., a procedure, function or method). Component testing concentrates on testing each subsystem individually. Integration testing checks the working order for a set of correct components interacting with each other. To check if the system works correctly as a whole, system testing is used.

In addition to identifying which abstract layer of the SUT needs to be tested, deciding which aspects of the SUT are to be fully checked is equally important. Several testing types have been proposed that cover different aspects of the SUT, such as stress, robustness, performance, reliability, and conformance. Stress testing checks if the SUT has consistent behaviour under a heavy load. Robustness testing involves investigating the reaction of the SUT under unexpected circumstances such as inputs being out of range or hardware failure. Performance testing checks the execution time of tasks performed by the SUT. Reliability testing ensures that the SUT is almost fault-free before its deployment. Finally, conformance testing aims at testing the functionality of the SUT to determine whether its behaviour conforms to that specified [29], [32].

The third axis in **Figure 2.1** shows two types of testing (white box and black-box) used according to the SUT visibility to the tester. White box testing is used to test the internal structure of the SUT whose algorithms and code are visible to the tester. Test cases are then designed using the information available about the SUT internal structure using different test selection methods. White box testing is supported by a Control Flow Graph (CFG) which graphically represents the code through its notations. As a result, test selection criteria can be complemented through the use of CFG. The oracle problem of white box testing concentrates on checking the correctness of SUT implemented behaviour at various levels such as unit-based or system-based. However, white box testing fails to check SUT behaviour according to a reference specification [29], [32].

On the other hand, black-box testing involves testing the functionality of the SUT according to a reference specification. The SUT internal structure (e.g., code) in black-box testing is not visible to the tester. The specification forms the source from which test cases are generated. Test cases are then sent to the SUT which emits output sequences. Several test selection strategies can be used in the case of black box testing such as adequacy criteria (e.g., state or transition coverage).

In contrast to white box testing, black-box testing is effective in testing SUT behaviour according to the specification but cannot guarantee whether SUT internal behaviour is correct [29], [32].

2.4 Model-Driven Architecture (MDA)

As software systems become increasingly complex, new paradigms are needed for their construction. One of these new paradigms is model-driven architecture (MDA), which already has a demonstrable impact in reducing time to market and improving product quality. This particular paradigm concerned with the introduction of rigorous models throughout the development process, enabling abstraction and automation.

The development of high-quality systems requires not only systematic development processes but also systematic test processes. Therefore, MDT is inspired by the philosophy of MDA [33].

As shown in **Figure 2.2**, platform-independent system design models (PIM) can be transformed into platform-specific system design models (PSM). While PIMs focus on describing the pure functioning of a system independently from potential platforms that may be used to realize and execute the system, the relating PSMs contain a lot of information on the underlying platform. In another transformation step, system code may be derived from the PSM. Certainly, the completeness of the code depends on the completeness of the system design model.

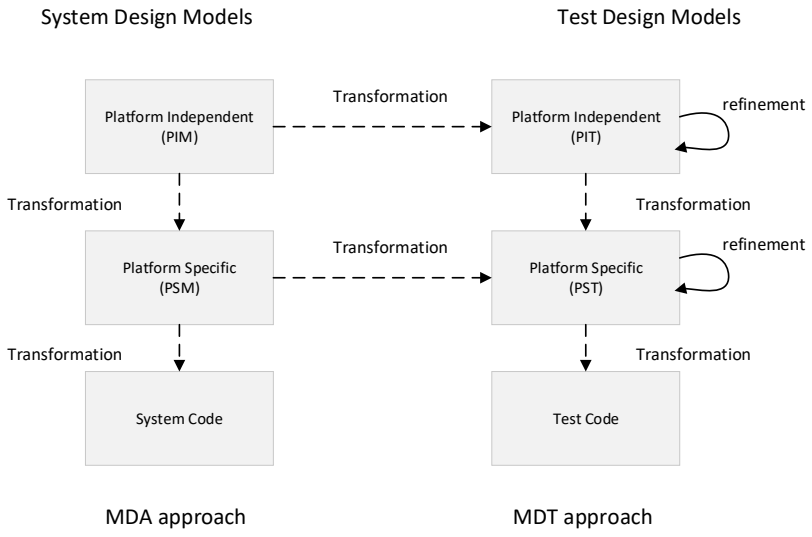


Figure 2.2: Model-driven architecture paradigm

The same abstraction in terms of platform-independent, platform-specific modeling and system code generation can be applied to test design models.

Furthermore, test design models might be transformed from system design models directly. This enables the early integration of test development into the overall development process. Once the system design model is defined at the PIM level, a platform-independent test design model (PIT) can be derived. This model can be transformed either directly to test code or to a platform-specific test design model (PST) [34]. The same transformation technology can be used for deriving PSTs from the PSM. After each transformation step, the test design model can be refined and enriched with test specific properties. Although the transformed test design model may already contain static and dynamic aspects, the behavior has to be completed to cover unexpected system behavior as well. Also, test issues such as e.g. test control and deployment information have to be manually added to the test design model. At last, the test design model can be finally transformed into executable test code from either PST or PIT.

2.5 Model-Based Testing (MBT)

MBT relates to a process of test generation from models of/related to a SUT by applying several sophisticated methods. Several authors such as Utting [35] and Kamga, Hermann, and Joshi [36] define MBT as testing in which test cases are derived in their entirety or in part from a model that describes some aspects of the SUT based on selected criteria. In MBT which has the highest focus, informal requirements of the system are the base for developing a test model which is a behavioral model of the system. This test model is used to automatically generate test cases [37]. One problem in this area is that the generated tests from the model cannot be executed directly against SUT because they are at the same level of abstraction as the model. The automation of an MBT approach depends on three key elements: (i) the model used for the software behavior description, (ii) the test-generation algorithm (criteria), and (iii) tools that generate supporting infrastructure for the tests. The authors in [38], [39] have worked on testing including MBT and are investigating new MBT and automation solutions. Others in [40], [41], and [42] describe MBT related surveys on test data generation techniques, supporting tools, and test case generation approaches respectively. However, no formal survey on the analysis of MBT approaches have been found. To our knowledge, this is the first scientific survey paper on MBT approaches using a formal methodology – Systematic Review [43]. Other important

characteristics are testing levels of MBT, automation levels, and complexity of non-automated steps. The process of model-based testing can cover various testing activities at different dimensions as depicted in **Figure 2.3.**

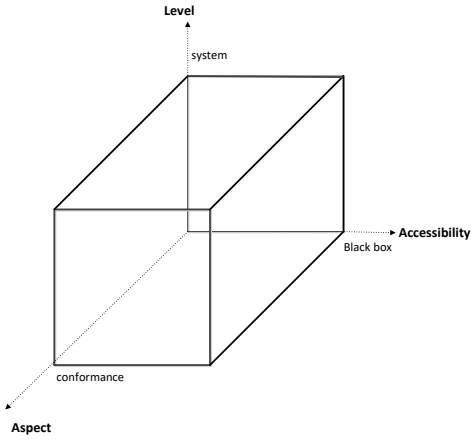


Figure 2.3: MBT with relation to other testing types (Briones, 2007)

MBT is considered as a form of black-box testing since test cases are generated from the specification model without accessing the implementation. MBT can also be used at any software level (e.g., component, integration or system). However, testing at the system level can be considered the most common use for MBT. Moreover, using MBT for testing other software aspects such as robustness is possible. The rationale for adopting MBT, however, is to examine conformance between SUT functional behaviour and a reference specification model.

2.6 Model Transformation

Model composition approaches automate the composition between heterogeneous models by relying on a matching and a merging operator [44]. Model-driven approaches move development focus from third-generation programming language code to models. The objective is to increase productivity and reduce time to market by enabling development and using concepts closer to the problem domain at hand, rather than those offered by programming languages. Model-driven development's key challenge is to transform these higher-level models to platform-specific models that tools can use to generate code[45]. We can use models not only horizontally to describe different system aspects but also vertically, to be refined from higher to lower levels of abstraction. At the lowest level, models use implementation technology concepts. Working with

multiple, interrelated models requires significant effort to ensure their overall consistency. In addition to vertical and horizontal model synchronization, we can significantly reduce the burden of other activities, such as reverse engineering, view generation, application of patterns, or refactoring, through automation. Many of these activities are performed as automated processes that take one or more source models as input and produce one or more target models as output while following a set of transformation rules. We refer to this process as model transformation.

Here, we give some model-driven engineering definitions, analyze current approaches to model transformation, and present the different design features for model transformation that can be used by modeling and design tools to automate tasks, thus significantly improving development productivity and quality.

2.6.1 Definition

Before classifying model transformation techniques, one should understand some model-driven engineering definitions [46], [47].

- **Definition 1 System Model:** A system model is an abstract representation of certain aspects of the SUT. A typical application of the system model in the MBT process leverages its behavioral description for the derivation of tests.
- **Definition 2 Model Transformation:** transformation is the automatic generation of a target model from a source model, according to a transformation definition.
- **Definition 3 Transformation Rule:** is a description of how one or more constructs in the source language, left-hand side (LHS), can be transformed into one or more constructs in the target language right-hand side (RHS).
- **Definition 4 Technical space:** is a model management framework containing concepts, tools, mechanisms, techniques, languages, and formalisms associated with a particular technology.
- **Definition 5 Endogenous transformation:** is the transformation between models expressed in the same language.
- **Definition 6 Exogenous transformation:** is transformation between models expressed using different languages.

— **Definition 7 Horizontal transformation:** is a transformation where the source and target models reside at the same abstraction level.

— **Definition 8 Vertical transformation:** is a transformation where the source and target models reside at different abstraction levels.

2.6.2 Model Transformation Categories

For the model-driven software development vision to become reality, tools must support this automation [48]. Development tools should let users not only apply predefined model transformations but also define their own. Performing a model transformation requires a clear understanding of the abstract syntax and semantics of both the source and target. Metamodeling is a common technique for defining the abstract syntax of models and the interrelationships between model elements. For visual modeling languages, there are several advantages in basing a tool's implementation on the language's metamodel. Such tools offer users three different architectural approaches for defining transformations [48]:

— **Direct model manipulation:** access to an internal model representation and the ability to manipulate the representation using a set of procedural APIs.

One advantage of the direct-model manipulation approach is that the language used to access and manipulate the exposed APIs is commonly a general-purpose language such as Visual Basic or Java, so the developers need little or no extra training to write transformations. Furthermore, developers are generally more comfortable with encoding complicated (transformation) algorithms in procedural languages. Examples are Rational Rose, which offers a version of VB with a set of APIs to manipulate models, and Rational XDE, which exposes an extensive set of APIs to its model server that can be used from Java, VB, or C#. A disadvantage is that the APIs usually restrict the kind of transformations that can be performed. Also, because the programming languages are general-purpose, they lack suitable high-level abstractions for specifying transformations. Consequently, transformations can be hard to write, comprehend, and maintain. One proposal that promises to raise the level of abstraction of operations on UML models is UML's action language. This special-purpose language has been proposed as a way to procedurally define UML transformations and manipulate UML models. However, the language still suffers, although less chronically, from

a lack of high-level abstractions for dealing with model transformations—for example, transformation composition.

— **Intermediate representation:** exporting of the model in a standard form, typically XML, so an external tool can transform it.

For the intermediate-representation approach, many UML tools can export and import models to and from XMI, which is an XML-based standard for the interchange of UML models. Because a model is externalized into XML, it is possible to use existing XML tools, such as XSLT, to perform model transformations. Even though XSLT was defined specifically for describing transformations, it is nevertheless tightly coupled to the XML that it manipulates. Consequently, it requires experience and considerable effort to define even simple model transformations in XSLT. Another disadvantage of the approach is that transformations are performed in batch mode, which has two important consequences. First, transformations are hard to perform in an interactive dialogue with the user. Second, the tool still needs to reactively manage the synchronization between models after changes. For example, a long and complex transformation performed outside of the tools might be rejected because of the violation of cross model integrity constraints.

— **Transformation language support:** a language that provides a set of constructs for explicitly expressing, composing, and applying transformations.

Transformation language support, as the name suggests, provides a specific language for describing model transformations. It offers the most potential of the three approaches because the language can be tailored for that purpose. In this context, you can use many languages to specify and execute model transformations, some of which offer visual constructs. These languages are either declarative, procedural, or a combination of both. For example, in [49] the author proposes a graphical language for describing model transformations that are principally procedural but also offers some declarative features. A tool that generates C++ code from the specification supports the approach. One limitation is its underlying assumption that you can easily express your choice of source model elements for the transformation in a general-purpose programming language, that is, C++. The Rational XDE's pattern mechanism is a commercial example of a specialized transformation language. This mechanism is built on top of XDE's model server API, so XDE supports both the direct model manipulation and transformation language support classifications.

XDE transformations are defined as model templates called patterns, which could contain parameters and arbitrary procedural code written in Java, VB, or C#. You can invoke patterns using a set of predefined callbacks; this effectively means you can make arbitrary "manual" model changes. The key drawback of the XDE's pattern engine is that it provides limited capability to compose patterns. Another general approach is to treat UML models as graphs. Applying graph rewriting rules help identify graph transformations. A rule consists of a graph to match, commonly referred to as LHS, and a replacement graph, commonly referred to as RHS. If a match is found for the LHS graph, then the rule is fired. Consequently, the RHS graph replaces the matched subgraph of the graph under transformation. The author in [50] has also proposed the use of rewriting rules for UML model transformation in the context of logic languages.

Beyond automating transformation execution, tools could suggest which model transformations a user might appropriately apply in a given context. In the next section, we present different design choices for model transformation.

2.6.3 Design Features for Model Transformation

In [47], the authors proposed a possible taxonomy for the classification of several existing and proposed model transformation approaches. The taxonomy is described with a feature model (Appendix E) that makes the different design choices for model transformations explicit. Each of the following subsections elaborates on one major area of variation from a feature model by describing the different choices and providing examples of approaches supporting a given feature.

2.6.3.1 Transformation Rules

As mentioned in the definition, a transformation rule consists of two parts: an LHS and an RHS. The LHS accesses the source model, whereas the RHS expands in the target model. Both LHS and RHS can be represented using any mixture of the following:

Variables: Variables hold elements from the source and/or target models (or some intermediate elements). They are sometimes referred to as metavariables to distinguish them from variables that may be part of the transformed model (e.g., Java variables in transformed Java programs).

Patterns: Patterns are model fragments with zero or more variables. We can have string, term, and graph patterns. String patterns are used in textual templates. Model-to-model transformations usually use term or graph patterns. Patterns can be represented using the abstract or concrete syntax of the corresponding source or target model language, and the syntax can be textual and/or graphical.

Logic: Logic expresses computations and constraints on model elements. Logic may be non-executable or executable. Non-executable logic is used to specify a relationship between models. Executable logic can take a declarative or imperative form. Examples of the declarative form include object constraint language queries (OCL)-queries [51] to retrieve elements from the source model (e.g., XDE) [52] and the implicit creation of target elements through constraints. Imperative logic has often the form of programming language code calling repository APIs to manipulate models directly. For instance, the Java Metadata Interface [53] provides a Java API to access models in a MOF repository [54]. In the context of the QVT [55] standardization effort, the UML Action Semantic [56] can be used to specify imperative logic in a form that can be automatically mapped to different programming languages.

Both variables and patterns can be untyped, syntactically typed, or semantically typed. In the case of syntactic typing, a variable is associated with a metamodel element whose instances it can hold. Semantic typing allows for stronger properties to be asserted.

Four other aspects of transformation rules are:

i. Syntactic Separation: The RHS and LHS may or may not be syntactically separated. In other words, the rule syntax may specifically mark RHS and LHS as such (as in classical rewrite rules), or there might be no syntactic distinction (as in a transformation rule implemented as a Java program.

ii. Bidirectionality: A rule may be executable in both directions.

iii. Rule parameterization: Transformation rules may have additional control parameters allowing configuration and tuning.

iv. Intermediate structures: Some approaches e.g., VIsual Automated model TRAnsformations (VIATRA) and Graph Rewriting and Transformation Language

(GreAT) require the construction of intermediate model structures. This is particularly relevant when the model transformation happens in-place within a model.

2.6.3.2 Rule Application Scoping

Rule application scoping allows a transformation to restrict the parts of a model that participate in the transformation. Some approaches support flexible source model scoping using graphical languages such as Rational XDE [52] and GReAt where a scope smaller than the entire source model can be set. The latter can be important for performance reasons. The target scope is the scope of the target model, in which the RHS will be expanded (e.g., XDE).

2.6.3.3 Relationship between Source and Target

Some approaches mandate the creation of a new target model that has to be separate from the source (e.g., [57]). In some other approaches, source and target are always the same model, i.e., they only support in-place updates (e.g., VIsual Automated model TRAnsformations (VIATRA), GreAT). Yet other approaches (e.g., XDE) allow the target model to be a new model or an existing one, which could be the source model. The latter implies an in-place update. Furthermore, an approach could allow a destructive update of the existing target or update by extension only, i.e., where existing model elements cannot be removed. Approaches using non-deterministic selection and fixpoint iteration scheduling may restrict in-place updates to extension in order to ensure termination (e.g., VIATRA).

2.6.3.4 Rule Application Strategy

A rule needs to be applied to a specific location within its source scope. Since there may be more than one match for a rule within a given source scope, we need an application strategy. The strategy could be deterministic, non-deterministic or even interactive. For example, a deterministic strategy could exploit some standard traversal strategy (such as depth-first) over the containment hierarchy in the source.

Stratego [58] is an example of a term rewriting language with rich mechanisms to express traversal in tree structures. Examples of non-deterministic strategies include one-point application, where a rule is applied to one non-deterministically selected location, and concurrent application, where one rule is applied concurrently to all OOPSLA’03 Workshop [47] on

Generative Techniques in the Context of Model-Driven Architecture matching locations in the source (e.g., VIATRA). Sometimes, rule application is determined interactively (e.g. XDE).

The target location for a rule is usually deterministic. In the case of an in-place update, the source location becomes the target location (e.g. VIATRA or GreAT). In an approach with separate source and target models, traceability links can be used to determine the target (e.g. [57]): A rule may follow the traceability link to some target element that was created by some other rule and use the element as its target.

2.6.3.5 Rule Scheduling

Scheduling mechanisms determine the order in which individual rules are applied. The scheduling mechanism can vary in four main areas:

Form: The scheduling aspect can be expressed implicitly or explicitly. Implicit scheduling implies that the user has no explicit control on the scheduling algorithm defined by the tool (e.g., BOTL and OptimalJ [59]). The only way a user can influence the system-defined scheduling algorithm is by designing the patterns and logic of the rules to guarantee certain execution orders. For example, a given rule could check for some information that only some other rule would produce. Explicit scheduling has dedicated constructs to explicitly control the execution order. Explicit scheduling could be internal or external. In external scheduling, there is a clear separation between the rules and the scheduling logic (e.g., in VIATRA, rule scheduling is provided by an external finite state machine). In contrast, internal scheduling would be a mechanism allowing a transformation rule to directly invoke other rules.

Rule selection: Rules can be selected by an explicit condition (e.g. Jamda). Some approaches allow non-deterministic choices (e.g. BOTL). Alternatively, a conflict resolution mechanism based on priorities could be provided (although none of the investigated approaches implement conflict resolution). Interactive rule selection is also possible (e.g. XDE).

Rule iteration: Rule iteration mechanisms include recursion, looping, and fixpoint iteration (i.e., repeated application until no changes detected).

Phasing: The transformation process may be organized into several phases, where each phase has a specific purpose and only certain rules can be invoked in a given phase. For example, structure-oriented approaches such as Optimal have a separate phase to create the containment hierarchy of the target model and a separate phase to set the attributes and references in the target.

2.6.3.6 Rule Organization

Rule organization is concerned with composing and structuring multiple transformation rules. We consider three areas of variation in this context:

Modularity mechanisms: Some approaches allow packaging rules into modules (e.g., [60] and VIATRA). A module can import another module to access its content.

Reuse mechanisms: Reuse mechanisms offer a way to define a rule based on one or more other rules. In general, scheduling mechanisms can be used to define composite transformation rules; however, some approaches offer dedicated reuse mechanisms such as inheritance between rules (e.g. rule inheritance in [60], derivation in [61], extension in [57], specialization in [62]), inheritance between modules (e.g., unit inheritance in [60]), and logical composition (e.g. [62]).

Organizational structure: Rules may be organized according to the structure of the source language (as in attribute grammars, where actions are attached to the elements of the source language) or the target language, or they may have their independent organization. An example of the organization according to the structure of the target is. In this approach, there is one rule for each target element type and the rules are nested according to the containment hierarchy in the target metamodel. For example, if the target language has a package construct in which classes can be nested, the rule for creating packages will contain the rule for creating classes (which will contain rules for creating attributes and methods, etc.).

2.6.3.7 Traceability Links

Transformations may record links between their source and target elements. These links can be useful in performing impact analysis (i.e., analyzing how changing one model would affect other related models), synchronization between models, model-based debugging (i.e., mapping the stepwise execution of implementation back to its high-level model), and determining the target of

a transformation. Some approaches provide dedicated support for traceability (e.g., [61]), while others expect the user to encode traceability using the same mechanisms as for adding any other kinds of links in models (e.g., VIATRA, GreAT). Some approaches with dedicated support for traceability require developers to manually encode the creation of traceability links in the transformation rules, while others create traceability links automatically (e.g., [61]). In the case of automated support, the approach may still provide some control over how many traceability links get created (to limit the amount of traceability data). Finally, there is the choice of location where the links are stored, e.g., in the source and/or target, or separately. A preferable approach is to store a unique identifier in each model element and store the traceability information separate from the source and target.

2.6.3.8 Directionality

Transformations may be unidirectional or bidirectional. Unidirectional transformations can be executed in one direction only, in which case a target model is computed (or updated) based on a source model. Bidirectional transformations can be executed in both directions, which is useful in the context of synchronization between models. Bidirectional transformations can be achieved using bidirectional rules or by defining two separate complementary unidirectional rules, one for each direction.

Transformation rules are usually designed to have a functional character: given some input in the source model, they produce a concrete result in the target model. A declarative rule (i.e., one that only uses declarative logic and/or patterns) can often be applied in the inverse direction, too. However, since different inputs may lead to the same output, the inverse of a rule may not be a function. In this case, the inversion could enumerate several possible solutions (this could theoretically be infinite), or just establish part of the result concretely (because the part could be the same for all solutions) and use variables, defaults, or values already present in the output for the other parts. The invertibility of a transformation depends not only on the invertibility of the transformation rules but also on the invertibility of the scheduling logic. Inverting a set of rules may fail to produce any result due to non-termination. Most of the investigated approaches do not provide for bidirectionality. Notable exceptions are [62], [63], [64]. The latter does not provide for general bidirectionality. Instead, a transformation can be described at different levels of abstraction, where one level is invertible and another is not.

2.6.4 Model Transformation from UCM

UCM scenario notation can be used in the process of generating requirement-directed test suites. There are challenges to generate test cases from UCM models as they emphasize behavior rather than data, and they also abstract from detailed communication mechanisms. Therefore, UCM models are inappropriate for the derivation of implementation-level test cases. However, deriving test goals from UCM models can help improve UCM-based testing.

Several approaches for deriving test goals from UCM models exist in the literature. We distinguish three main approaches based on: (1) testing patterns, (2) UCM scenario definitions, and (3) transformations to formal specifications (e.g., in LOTOS).

1) **Testing Based on UCM Testing Patterns**:

In Amyot Thesis [65], testing patterns are developed that target the coverage of scenarios described in terms of UCM. These patterns aim to cover functional scenarios at various levels of completeness: The rationale is that covering UCM paths leads to the coverage of the associated events and responsibilities (and of their relative ordering) forming the requirements scenarios. This approach helps engineers make informed decisions about the level of coverage they want at a given point in a UCM model. However, this process is entirely manual.

2) **Testing Based on UCM Scenario Definitions**

An instance of a UCM scenario can be extracted from a UCM model given a scenario definition, see metamodel in Appendix D, and a path traversal algorithm allowing for the semi-automatic generation of test goals. The first algorithm was proposed by Miga et al. and prototyped in UCMNAV [66]. It was used to support the understanding of complex UCM models by highlighting the paths traversed according to the scenario definition. It was then extended to generate a Message Sequence Chart (MSC) representing the scenario linearly.

A new implementation of the traversal algorithm in UCMNAV was performed by Amyot [67] which decouples the result of the traversal (output in XML) from specific representations such as MSCs.

Amyot et al. [68], [69] developed a tool (UCMEXPORTER) that takes the resulting XML scenarios as input and converts them to MSCs (in Z.120 phrase representation [70]) or UML 1.5 sequence diagrams (in XMI format [71]), with various options offered to the user. A prototype export filter that generates TTCN-3 test skeletons is also included.

Patrice et al. used the traversal mechanism in jUCMNav to generate test purposes in TDL. In their transformation, they flattened the UCM scenario model to several scenario definitions where each scenario element is mapped to TDL elements. The result is several independent instances of TDL metamodel serialized in the XMI interchange format. The transformation plug-in is packaged with jUCMNav version 5.5.0 and above. The resulting format suffers from the following problems; (1) no support for alternative behaviour, (2) no concrete TDL syntax or grammar, and (3) no TDL semantic, the TDL elements are displayed as partial-order trace.

Suitable scenario definitions still need to be provided manually, but then the generation of the test goal is automated, which is a significant advantage when the UCM model evolves. Scenario definitions have been used to explore various types of systems and to generate more detailed scenarios, with design level artifacts such as inter-component messages. He et al. [72] used MSC scenarios generated from a UCM model (via scenario definitions and UCMNAV) to explore the automated synthesis of SDL executable specifications. Klocwork's MSC2SDL, part of Telelogic Tau 4.5, was used to synthesize the specification. However, the authors have not explored the use of this specification to generate test cases in TTCN.

There is an obvious need to extend and validate the transformation much further, both in terms of technical correctness and usefulness. There exists a difference between a scenario, which is a (partial-order) trace in the UCM model, and a test case that can handle alternate test behavior, e.g., combinations of scenarios.

3) **Testing Based on UCM Transformations**

The third approach automates the generation of test goals by transforming a UCM model to the formal specification, e.g., in LOTOS.

Automated Generation of LOTOS Scenarios and TTCN Test Cases: To generate test goals, Charfi uses an exhaustive path traversal algorithm, adapted from Miga's original one, to traverse a UCM model augmented with key annotations in LOTOS [73]. This approach, prototyped in the

UCM2LOTOSTEST tool, produces an exhaustive collection of test goals described as partially-ordered sequences of LOTOS events. The tool does not consider the path data model, instead it maps condition labels to LOTOS events. The generation of test goals is automated, but the size of the resulting test suite grows very quickly as the UCM model becomes more complex. The test goals can be used, in combination with the specification and the TGV toolkit [74], to generate acceptance test cases in TTCN. Several minor modifications to the test goals were however required to be compatible with the requirements of TGV.

Automated Generation of LOTOS Specifications and Scenarios: Guan's thesis work [75] had a different purpose, which was the generation of scenarios in the form of MSCs from UCM models, in assistance to the process of producing precise and consistent documentation for telecommunications standards. The author developed an automatic translator from a substantial subset of the UCM notation to LOTOS. This work, prototyped in the UCM2LOTOSSPEC, improves greatly upon the approach suggested by Charfi where the LOTOS specification is produced manually because the specification can be re-generated each time the UCM model changes.

A companion tool based on the same principles, UCM2LOTOSSCENARIOS, can extract individual LOTOS scenarios or test goals from the UCM model. The generation of scenarios follows the structure of the UCM, in the sense that all possible paths in the UCM are traversed once. The generated test goals preserve the concurrency introduced in the UCM model (e.g., with AND-forks) using the LOTOS parallel operator (|||). The tool supports the generation of test goals from maps with loops and multiple start points. The LOTOS specification and the test goals so generated can be used to verify and validate UCM models. The research focuses on the translation algorithms and does not address the problems of scenario selection or elimination of unfeasible scenarios. Therefore, for complex UCMs, this method will produce large numbers of scenarios and many are likely to be unfeasible and will require a manual inspection to be detected.

2.7 Test Case Generation

A difficult part of software testing entails the generation of test cases which is one of the most intellectually demanding tasks and it is also of the most critical challenges since it can have a

strong impact on the effectiveness and efficiency of the whole testing process [76]; Test case generation is an important activity to cut down the cost of manual testing. It is no surprise that a great amount of research effort in the past few decades has been spent on automatic test case generation. As a result, a considerable number of different techniques for test case generation have been advanced and rigorously investigated.

In general, test cases are generated from several types of software artifacts. The types of artifacts that have been used as the reference input to the generation of test cases include: the program structure and/or source code; the software specifications and/or design models; information about the input/output data space, and information dynamically obtained from program execution. There are several techniques for test case generation [42] such as MBT techniques, random approaches, specification-based techniques, source code-based techniques, NL, web application and combined.

2.7.1 MBT Technique

MBT techniques are used to generate test cases from models like UML diagrams [77], [78], [79]. Many diagrams are used in generating a set of test cases, such as use case diagram, activity diagram, and statechart diagram. The literature shows that UML diagrammatic technique is the most widely used in the software design phase. Several approaches for generating test cases from different UML diagrams are proposed.

- In [80], the authors proposed an approach that links the requirement process with the testing process through a use case model. The approach creates system test cases based on two types of models: (1) UML use case models that describe the system requirements from test designers' point of view; and (2) various forms of MBT. The approach requires additional behavioral modeling such as activity diagram, sequence diagram, and class diagram models. The approach focuses on data flows that require manual intervention by test designers to annotate UML diagrams with additional test data such as coverage requirements, constraints, and preconditions.

- In [81], a model-driven process is proposed to generate automatically both formal models and test cases from the same UML model of the system under verification and validation and model transformation. The approach is applied to a railway control system that

features all the characteristics of a complex ES. The approach is based on formal methods to reduce the overall assessment effort and to support the validation against both functional and non-functional requirements. However, the formal models are time-consuming and expensive to generate and are difficult to be used as a communication mechanism for non-technical personnel.

- The authors of [82] proposed an MDT approach for testing applications designed in a model-driven development context (MDE). Their work focuses on the separation of generating test cases and oracles, and the execution of these tests on different target platforms. However, the work considers a specific issue and explicitly addresses the problem of test generation in MDE context.

- In [83], the authors propose a methodology TOTEM for system testing to derive system test requirements from early UML artifacts such as use case, class, and sequence diagrams. The authors propose to express the sequential constraints of the use cases with an extended activity diagram that are transformed into a weighted graph. The regular expressions that correspond to use case sequences are extracted from the weighted graph. The derivation of test artifacts from test requirements is delayed till the low-level design becomes complete, and when detailed information becomes available regarding application domain and solution domain classes.

- The authors of [84] propose to use restricted natural language for the specification of use cases. The use cases are mapped to a formal model (FSM) and test scenarios are generated by traversing the FSM based on coverage criteria. In this approach, there is a substantial overhead for diagram creation and modification of the use case description to the restricted natural language format.

- Another important approach to generate test cases from use cases is presented in [85]. The approach generates test cases in two phases. In the first phase, the approach describes system requirements via use case diagram, scenario, and contracts. Each use case is enhanced with contracts that are expressed in first-order logical expression to specify the preconditions and post-conditions. Next, the enhanced use cases are transformed to test objectives using a transition system known as Use Case Transition System (UCTS) that can represent all valid sequences of the use case. In the second phase, the test objectives

are transformed to test scenarios. Sequence diagrams are attached as additional artifacts to obtain sequences of message calls on the SUT. The approach requires working with various UML diagrams and formal methods.

- The authors of [86], have explored the automated generation of TDL Test Descriptions from requirements expressed as UCM scenario models using the jUCMNav tool. This transformation enables the exploration of model-based testing where the use of TDL models simplifies the generation of tests in various languages such as TTCN-3. The authors determined the basic differences between scenarios and test cases in the handling of alternative paths that result from UCM alternatives. They concluded that the use of scenarios for test case generation is feasible, but requires either a different traversal mechanism with a different scenario metamodel or post-processing of scenarios to merge those that constitute alternate test behaviors.
- In [87], the authors introduced an automatic test generation approach that provides more natural and standardized ways of writing requirements using document templates. These templates are extended to allow include and extension relations between use cases and to include data elements as user-defined types, variables, and parameters. The approach uses the use case templates that capture control flow, state, input and output as source for the generation of formal models. Unfortunately, it only generates non-ETCs.

The major advantages of model-based are that shifting the testing activities to an earlier part of the software development process and generating test cases that are independent of any particular implementation of the design [88]. The following paragraphs describe existing specification-based techniques that have been proposed since 2000.

2.7.2 Specification-Based Technique

Specification-based techniques are methods to generate a set of test cases from specification documents such as a formal requirements specification (Cunning and Rozenblit, 1999; Tran, 2001; Rayadurgam and Heimdahl, 2001a; Nilsson et al., 2006; Tsai et al., 2005), Z-specification (Huaikou and Ling, 2000; Jia and Liu, 2002; Jia et al., 2003) and Object Constraint Language (OCL) specification (Antonio et al., 2006). The drawbacks of the specification-based technique with formal methods are: (1) the difficulty of conducting formal analysis and the perceived or

actual payoff in the project budget and (2) there is greater manual effort or processes in generating test cases, compared with techniques involving automatic generation processes. The following describes existing specification-based techniques that have been proposed since 1997.

- Kancherla (1997) used a form of specification-based testing that employs the use of an automated theorem prover to generate test cases. A similar approach was developed using a model checker on state-intensive systems. The method applies to systems with functional rather than state-based behaviors. The approach allows for the use of incomplete specifications to aid in the generation of tests for potential failure cases. He suggested a new method of testing software based on the formal specification. He used the Prototype Verification System (PVS) and its in-built theorem prover to derive test cases corresponding to the properties stated in the requirements.

- Cunning and Rozenblit (1999) were interested in the model-based co-design of real-time ESs. It relies on system models at increasing levels of fidelity to explore design alternatives and to evaluate the correctness of these designs. As a result, the tests that they desire should cover all system requirements in order to determine if all requirements have been implemented in the design. The set of generated tests is maintained and applied to system models of increasing fidelity and to the system prototype to verify the consistency between models and physical realizations. In the co-design method, test cases are used to validate system models and prototypes against the requirements specification. In the study, they presented continuing research toward automatic generation of test cases from requirements specifications for event-oriented, real-time ESs. They used a heuristic algorithm to the automatically generate test cases in their works. The heuristic algorithm uses the greedy search method followed by a distance-based search if needed. The algorithm with pseudo code is addressed (Cunning and Rozenblit, 1999).

2.7.3 NL Technique

Most of the software industry works with requirements expressed in NL. Several approaches are proposed

- The approach in [78] presents a method (SCENT) to create scenarios from NL and formalize them in state charts. An annotation technique is then used to enrich the statecharts with helpful information. A path traversal algorithm is employed in the

statecharts to determine concrete test cases. The test suite is further enhanced by generating test cases from dependency charts that are modeled from dependencies between scenarios. SCENT requires two different representations of the scenarios, which makes it rather costly in terms of the testing effort.

- The approach in [89] generates test cases based on NL requirements' specifications using a tool. The tool models the NL requirements into UML activity diagrams to support automated testing. This approach requires using a scenario language [90] that references relevant words from the application with lexicon symbols.

In the conclusion, there are three major sources used to generate test cases, which are: (1) requirements expressed in UML diagrams, (2) formal requirement specifications and (3) requirements expressed in natural language.

2.8 Traceability

The largest part of traceability research so far has been done in the last two decades by the requirements engineering community [91]. Over the past years, it has gained in importance, and traceability topics have become subject to research in many other areas of software development. One of these areas is model-driven development (MDD), an area where parts of the software development process are executed automatically using model transformations [91], [92].

2.8.1 Requirement Traceability

In the domain of requirements engineering, the term *traceability* is usually defined as the ability to follow the traces (or, in short, to *trace*) to and from requirements. Two common definitions of requirements traceability are given by Pinheiro [93] as the ability to define, capture, and follow the traces left by requirements on other elements of the software development environment and the traces left by those elements on requirements. And by Gotel and Finkelstein [94] as the ability to describe and follow the life of a requirement, in both a forwards and backward direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and periods of on-going refinement and iteration in any of these phases).

2.8.2 Traceability in MDD

In the context of MDD, traces partially fulfill the same purpose as in requirements engineering because in many tasks, MDD is simply an automation of software engineering. The special characteristic of MDD is the usage of models and automated transformations. So, the artifacts under study are mainly (intermediate) models. This context influences the definitions and semantics of the terms known from requirements traceability and software engineering in general. In addition, the “MDD way” to define terms is often to simply define models and metamodels in which they occur. This is why most publications either do not refer to an explicit definition of traceability at all or only refer to the general IEEE definition cited above. Also, since traceability cannot be modeled intuitively, most definitions refer to traceability links. An example of a model-like definition for the term traceability is the rather technical and narrow definition that is given by the OMG [95]: A trace records a link between a group of objects from the input models and a group of objects in the output models. This link is associated with an element from the model transformation specification that relates the groups concerned. A commonality between MBT and traceability is required to manage relationships among artifacts. Relationship management should assist conception, persistence, preservation, and destruction of meaningful relationships across software artifacts [96].

2.8.3 Alignment of Requirements Traceability and Testing

In a recent study [97], the authors highlight the importance of aligning the activities of requirement traceability to testing to improve system quality and project cost. The study concluded that organizations are becoming more interested in linking requirements and testing, but often the link is not provided and there is a gap between them.

Several researches in the study were identified that focus on the alignment of requirements specification and testing. In MBT-based approaches, the generated test data cannot be executed directly on SUT because they are at the same level of abstraction as the model. In formal approaches, representing requirements in a formal language is time-consuming and requires expertise.

Traceability links can be visualized in a traceability matrix, as cross-references in the table-view or in a model- or graph-like diagram. In order to support relationship management among

requirements and test cases, several approaches use the traceability matrix and MBT to represent the relationships that exist.

2.8.4 Matrix Approach

A traceability matrix is a two-dimensional grid that represents traceability links that exist between two sets of artifacts, such as requirements, design elements, etc. The rows and columns of the grid are associated with the artifacts, and marks at the intersections represent the existence of a link.

While early forms of traceability matrices only provided support for a single type of mark representing the existence or non-existence of a link between two artifacts, traceability matrices today can be enhanced to include additional information about artifacts and links [93]. For example, an artifact in the matrix is usually referenced using an identifier but modern user interfaces can provide popup windows directly showing an artifact's meta-information or content if needed. Furthermore, link types or other information could be encoded using different colors or symbols [98].

- One example of a semi-automatic solution [99] creates a traceability matrix from requirements to test cases during the test generation process. The formal models are annotated with requirements identifiers. When the test cases are generated from the models, the identifiers are used to create the traceability matrix relating requirement identifiers to test cases identifiers.

- Spanoudakis and Zisman [96] also provide a matrix containing pairs of artifacts and traceability link classes. This matrix gives an overview of which traceability links can connect with which artifacts according to the literature. A similar list has also been created by Espinoza et al. [100].

- A hierarchical classification has been created by Dahlstedt and Persson [101]. They base their classification on the first level of structural, constrain, and cost/value interdependency types. According to their classification, structural types, such as refined-to or similar- to denote structural cross-references.

2.8.5 MBT Approach

Automated MBT approaches exploit two types of relationships; (1) implicit relationships that are embedded in the tool's algorithms and models, (2) explicit relationships that are either automatically created and made explicit by the tool, or created by the users. Several approaches [102], [103] use implicit relationships to support test generation, execution and evaluation; while others [104] use implicit relationships to support regression testing. Further approaches use explicit relationships to support test generation [105], test execution and evaluation [106], [107] [108], or coverage analysis. In MDD, traceability links are often expressed as part of a model, and even in the requirements domain, traceability schemes are usually described as metamodels.

- N aslavsky et al. [109] use one kind of behavioral UML model for test generation. A control-flow representation is used along with domain analysis of the parameters of the sequence diagram.
- Basanieri et al. [105] use a tool (COW_SUITE) that loads UML models to create explicit relationships as edges in hierarchical trees among them.
- The authors in [110] adopt the tool (AGEDIS) that uses explicit relationships created by the user to execute and evaluate the test scripts. The created relationships map abstract stimuli to method invocations, and abstract observations to value checking. The tool also expresses relationships between abstract test suites and test trace results during test execution. Manual coverage analysis is supported via visualization of test traces and the abstract test suite that generated them.
- In [107], the (AsmL) tool uses explicit relationships created by the user to execute and evaluate the abstract test scripts. The use of relationships in AsmL tool supports the parallel execution of the model and its implementation by relating them and comparing their states.
- Abbors et al. [111] present an approach for requirements traceability across an MBT process and the tools that are used for each phase. Some prior researches address requirement-based testing to facilitate traceability between requirements and testing.

- Arnold et al. propose a scenario-driven approach [112] that supports the traceability between generated and executed test cases, and the executions of an IUT. Their approach supports both FRs and NFRs.
- Goel et al. [113] propose a model-driven approach in which the strengths of both scenarios-based and state-based modeling styles are combined. Their tool makes it possible to trace from requirements to testing and vice versa in a round-trip engineering approach.
- Pfaller et al. propose [114] using different levels of abstraction in the development process to derive test cases and link them with the corresponding user requirements
- Boulanger and Dao propose an approach [115] in which RE is done in different phases of the V-model to facilitate requirements validation and traceability.
- Felderer et al. focus on model–driven testing of service-oriented systems in a test–driven manner [110]. They believe that Telling TestStories tool could support traceability between all kinds of modeling and system artifacts.
- Marelly et al. extend sequence charts (LSCs) with symbolic instances and symbolic variables [107] to reach linking requirements and testing.

2.8.6 Formal Approach

- Post et al. focus on translating requirements into scenario-based formal language which in turn could be linked to software verification [116].
- Bouquet et al. use a subset of UML 2.0 diagrams and Object Constraint Language (OCL) operators to formalize the expected system behavior [117]. The model is used for automatically generating executable test scripts.
- Kelleher and Simonss propose a new requirement modeling approach [118] in which use cases are replaced with use-case classes in UML 2.0. Use case classes are formal templates for describing rules on modeling requirements with instances. This replacement, together with utilizing explicit traceability links, facilitates bridging the gap between requirements and testing.

- Sabetta et al. discuss [119] that sometimes it might be needed to transform UML models into different analysis models which could each be used to verify (in a formal way) one kind of NFR. Some of these models are Petri nets, queueing networks, formal logic, etc. For this purpose, their abstraction approach can transform UML models into different kinds of analysis models in different formalisms.

- Hussain and Eschbach present a model-based safety analysis approach [120] that automatically composes formal models of the system and produces a fault tree that can be used to generate test cases for the software system. Therefore, test cases can be directly bound to the safety requirements and assure traceability between testing activity and safety requirements.

2.8.7 Meta-Model Approach

- Ibrahim et al. construct a meta-model with top-down and bottom-up traceability support [121]. The authors developed an approach that gathers traceability relations from different sources. Requirements and test cases are connected while analyzing system documentation. Test cases and methods are linked via test execution, where methods and classes are linked by static program analysis. The traceability approach provides some leverage. However, the bottom-up traceability provides less accuracy and requires more maintenance effort.

- Dubois et al. propose a meta-model called DARWIN4REQ which aims to keep the traceability link between three phases of requirement elicitation, design, and V&V of requirements [122]. The authors investigated strategies for requirements traceability based on models but focusing on subdomains of embedded systems.

2.8.8 Test Case Approach

- Nebut et al. concentrate on a guideline for automatic test case generation on ESs that are based on object-oriented concepts [85] The system requirements are described via use cases, contracts, and scenarios. If any other information for the requirements is needed, it is provided by different UML artifacts like sequence diagrams.

- Whalen et al. mention several problems of measuring the adequacy of black-box testing using executable artifacts [123]. They also present coverage metrics based on formal

high-level software requirements. Conrad et al. presented a test case generation strategy that has been in use in an automotive company [124].

- Siegl et al. are also interested in the automotive industry proposed Extended Automation Method (EXAM) for automatic generation of test cases, and the Timed Usage Model process for derivation of test cases from requirements [86].
- Riebisch and Hubner concentrate on the first step of test case generation [125]. In this step, their proposed method uses a description of the natural language and transforms it into an expression with formally defined syntax and semantics.

2.9 Summary of Literature Review

Testing embedded systems software has become a costly activity as these systems become more complex to fulfill rising needs. Testing processes should be both effective and efficient. An ideal testing process should begin with validated requirements and begin as early as possible so that requirements defects can be fixed before they propagate and become more difficult to address.

Among a range of testing activities, test case generation is one of the most intellectually demanding tasks and it is also of the most critical challenges since it can have a strong impact on the effectiveness and efficiency of the whole testing process.

Since traceability is mainly achieved by documenting different aspects of (usually manual) transformations of software development artifacts, MDD seems to be able to leverage traceability by automatically generating these documentations. However, traceability practices, in general, are far from mature, benefits are to a large part not conceived in the industry, and we are still standing at the beginning of an emerging discipline. A lot of research—both fundamental and applied—has still to be done. This is a challenge, not only because of the difficult research questions, but also because researchers in the field of traceability are usually part of very different larger research communities (such as requirements engineering, modeling, or program understanding), and there is only little communication between these communities.

Chapter 3 Domain Specific Languages (DSLs)

In the last few years, domain-specific language (DSL) has been getting more and more attention in the software industry. DSL is a small, usually declarative, language that offers expressive power focused on a particular problem domain. One of the main goals of DSLs is to enable the developer to define completely new languages that have more appropriate concepts for special domains. Furthermore, developers get the advantages of development activities on a higher level of abstraction. Languages are represented in different ways: by metamodels specified in some data modeling technique or by formal grammars. Although many DSLs have been designed and used over the years, the systematic study of DSLs has only started more recently.

In this Chapter, we introduce three DSLs that we used in our approach to (1) capture functional requirements in terms of causal scenarios, (2) describe the software ATCs as scenarios and (3) implement TCs and execute them against SUT.

3.1 Use Case Maps (UCM)

UCM: a visual notation for describing, in a high-level way, how the organizational structure of a complex system and the emergent behaviour of that system are intertwined. UCM [8] as part of the User Requirements Notation standard was suggested to represent the behaviour of a system as a visual use case, i.e. a scenario model. UCM is a scenario-based notation enabling the description and analysis of use cases and scenarios. It has been used to capture functional requirements in terms of causal scenarios composed of responsibilities that can be attached to underlying abstract components. UCM models have maps that contain any number of paths and components. The core notation of UCM has the following fundamental elements. *Paths* express causal sequences starting at *start points* and ending at *endpoints*, which respectively capture triggering and resulting conditions/events. Along a path, *responsibilities* describe the required activities to fulfill a scenario. Paths can be combined as alternatives with guarded *OR-forks* and merged with *OR-joins*, while *AND-forks* and *AND-joins* depict concurrency. Loops can be modeled implicitly with OR-joins and OR-forks. Joins and forks may be freely combined. *Waiting places* and *timers* denote locations on the path where the scenario stops until a condition is satisfied. UCM models can be decomposed using *stubs* (static or dynamic), that contain sub-

maps. *Components* are used to specify the structural aspects of a system. Map elements that reside inside a component are said to be bound to it. Components, which can be of different types (not shown here), can also contain sub-components, recursively. UCM models can be edited, analyzed and transformed with the jUCMNav tool [18]. One of its main features is a *UCM traversal mechanism* that takes as input a model and a *scenario definition* (start points triggered, and initial values assigned to the model variables used in OR-fork/timer/stub conditions) and produces as output a scenario that contains the UCM elements traversed. Generated scenarios are partial orders containing sequenced and concurrent responsibilities only; all conditions and alternatives have been resolved during the traversal. A scenario can be used to highlight the paths traversed on the visual model itself (e.g., in red or grey).

In **Figure 3.1**, we find a model with one map contains: a *Causal path* represented by a wiggly line, two *rectangular boxes* that represent components (Tester and SUT) four *responsibilities* bound to components along the path, and one highlighted scenario.

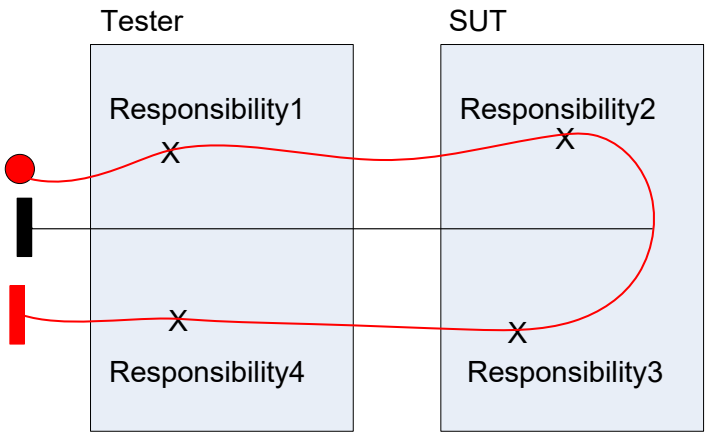


Figure 3.1: UCM core notation

The *responsibilities* elements in UCM are abstract and can represent actions or tasks to be performed by the components. The components themselves are also abstract and can represent software entities (objects, processes, network entities, etc.) as well as non-software entities (e.g. users, actors, processors). The concrete metamodel of the UCM notation is shown in Appendix A where the UCM quick reference guide is shown in Appendix F.

3.2 Test Description Language (TDL)

TDL: TDL is a standardized scenario-based approach proposed by the European Telecommunications Standards Institute (ETSI) to describe software test cases as scenarios. TDL is a new language created for specifying "formally defined test descriptions used as the starting point for test automation. It allows describing scenarios on a higher abstraction level than programming or scripting languages. Furthermore, TDL can be used as an intermediate representation of tests generated from other sources, e.g. simulators, test case generators, or logs from previous test runs." [126]. TDL is a general formal language for representing test descriptions which are used mainly for communication between stakeholders as the basis for implementing concrete tests. The TDL design is centered on three separate concepts: (1) The metamodel principle that expresses its abstract syntax; (2) Concrete Syntax, which is user-defined for different application domains; and (3) the TDL semantics that can be found in meta-model elements. The main TDL structure elements expressed in *italic* are shown in **Figure 3.2**.

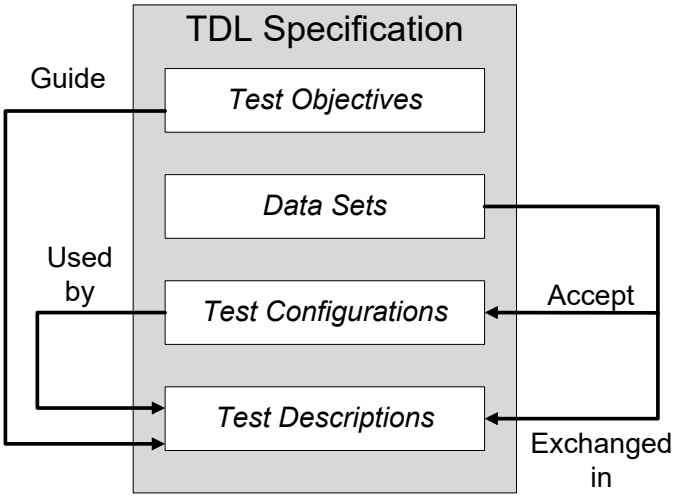


Figure 3.2: Major parts of a TDL specification

a) A *Test Objective* that states the reason for designing either a *Test Description* or a particular behaviour of a *Test Description*. It can be written as a simple text in natural language.

b) A set of typed *Data Sets* used in the interactions between components in a *Test Description*;

c) A *Test Configuration* consisting of at least one tester and at least one SUT component and connections among them reflecting the test environment.

d) A set of *Test Descriptions* to describe one or more test scenarios based on the interactions of data exchanged between the Tester and the SUTs. It also contains behavioral elements that operate on time. The control flow of a *Test Description* is expressed in terms of the composition of operations such as sequential, parallel, alternative, iterative, etc.

Using these major ingredients, a TDL specification is abstract in the following sense:

- Interactions between tester and SUT components of a test configuration are considered to be atomic and not detailed further. For example, an interaction can represent a message exchange, a remote function/procedure call, or a shared variable access.
- All behavioural elements within a test description are ordered unless it is specified otherwise. That is, there is an implicit synchronization mechanism assumed to exist between the components of a test configuration.
- The behaviour of a test description represents the expected, foreseen behaviour of a test scenario assuming an implicit test verdict mechanism if it is not specified otherwise. If the specified behaviour of a test description is executed, the 'pass' test verdict is assumed. Any deviation from this expected behaviour is considered to be a failure of the SUT, therefore the 'fail' verdict is assumed. There is a possibility for explicit verdict assignment if in a certain case there is a need to override this implicit verdict setting mechanism (e.g. to assign 'inconclusive' or any user-defined verdict values). However, there is no assumption about verdict arbitration, which is implementation-specific.
- The data exchanged via interactions and used in parameters of test descriptions are represented as name tuples without further details of their underlying semantics, which is implementation-specific.

A TDL specification represents a closed system of tester and SUT components. That is, each interaction of a test description refers to one source component and at least one target component that is part of the underlying test configuration a test description runs on. The actions of the actors (entities of the environment of the given test configuration) can be indicated informally.

Time in TDL is considered to be global and progresses in discrete quantities of arbitrary granularity. Progress in time is expressed as a monotonically increasing function. Time starts with the execution of an unreferenced ('base') test description. TDL can be extended with tool, application, or framework-specific information by use of annotations.

The TDL elements are explained with an example based on the Internet's Domain Name System (DNS) that aims at verifying that a DNS server can properly resolve hostnames to their corresponding IP addresses. The *Test Configuration* element that is composed of a set of two interacting components is shown in **Figure 3.3**.

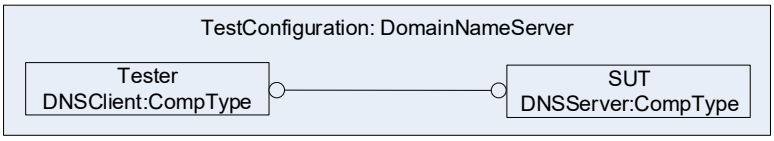


Figure 3.3: TDL Test Configuration element

The *Test Description* element represents the expected behaviour based on the *Test Objective* and expresses the test in terms of *Data Set instances* exchanged as shown in **Figure 3.4**.

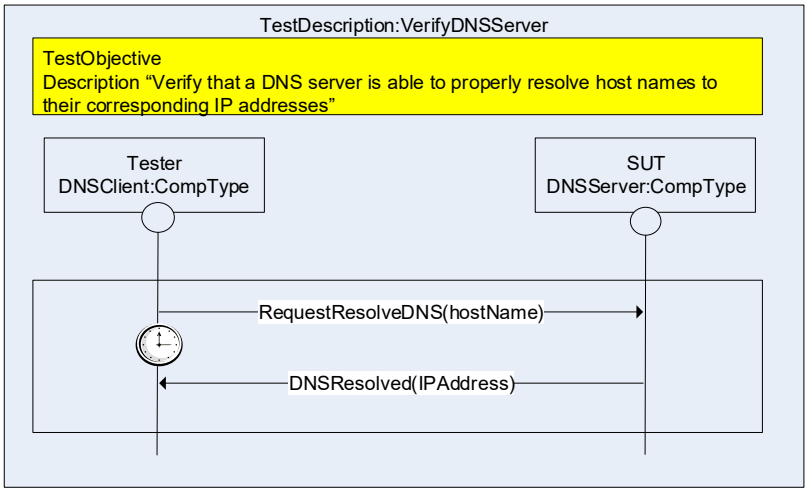


Figure 3.4: TDL Test Description element

Appendix B and Appendix C show the metamodel of *Test Configuration* and *Test Description*. Interested readers can refer to [126], [127] that discuss the application of TDL to several common application scenarios.

3.3 Testing and Test Control Notation (TTCN-3)

TTCN-3: a standard language for test specification that is widespread and well-established. The core language has to be transformed to a programming language such as Java, C, C++ or C#. There are number of commercial and non-commercial tools that provide supports to the language [128]. TTCN-3 is meant for specifying collections of test cases, Abstract Test Suites (ATS). To be able to execute the test cases within an ATS, a tool (compiler, interpreter) is required to transform the ATS into an executable test suite. In the following, TTCN-3 Core language is explored.

Module: the TTCN-3 language element called module corresponds to a compilation unit in traditional programming languages. It can be analyzed, compiled or interpreted, it may contain a single or several test cases, and it can be used as a library by other modules. Each module is divided into two parts, definitions part and control part, both of which are optional. The definitions part contains top-level definitions, such as type definitions, data (template) and constant definitions, port and component definitions, and function and test case definitions. The control part can be seen as the "main function" of the module and its purpose is to call the test cases defined in the part of the definitions. It contains the logic for executing the test cases in a certain order, it can apply execution time restrictions to the test cases, and it can use the definitions specified in the definitions part of the module to specify local variables. It is possible to specify parameters for a module, meaning that when a test case or the control part of the module is executed, it can read these parameters and behave according to them. The parameters are like module global constants, whose values are set at the start of the execution.

Test case: a test case can be seen as the main function of a single case, and of any other functionality executed in parallel with the test case. A test case is always executed within an entity called component, and it can call normal functions and altsteps to extend its behavior. The result of executing a test case is a verdict, which tells whether the system under test passed the test. A test case can be both a message- and a procedure-based.

Message-based testing consists of sending messages to the SUT, receiving messages from it, checking whether messages were not received in time, and checking whether the received messages are in the right order and that they contain the right values. Procedure-based testing

consists of calling functions of the SUT, receiving return values and exceptions, receiving function calls, and of passing function return values and raised exceptions to the SUT.

Components, Ports, and Test Configurations: the behavior of a single test case consists of executing functionality (test cases and functions) in one or more components. A component is a user-specified entity, which contains user-defined ports, via which the component can interact with other components and the SUT with message and procedure operations. In addition to the ports, the component may contain private variables and timers. The component itself does not specify any kind of behavior but it provides an environment for it. This means that one can start functionality in the component and this functionality can then use the ports, variables, and timers of the component. The functionality that can be started in the component can be either a test case or a function, see **Figure 3.5**.

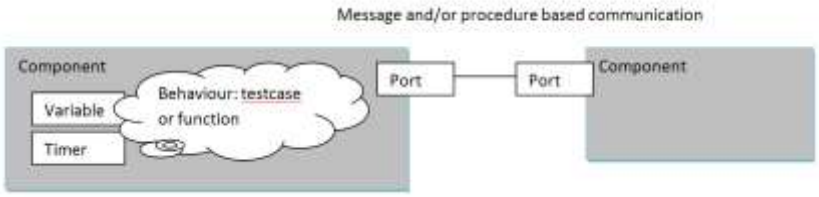


Figure 3.5: Model component

Verdict: every component that exists during a test case has a local object called verdict, which it can set (setverdict) based on how it experiences the behavior of the other components and the SUT. Components can also read their current verdict value (getverdict). The possible verdict values a component can set are none, pass, inconc, and fail. Once a component has set a value for its verdict, it can only "worsen" the verdict value.

Function: a normal function can have input parameters, output parameters, input-output parameters, and it can return a value. It is also possible to specify that the function can only be called or started within a component of a certain type, which makes the internal definitions of the component visible to the function (ports, timers, and variables).

Altstep: used for specifying action whose execution is triggered by some "receiving" event or operation, such as a timeout or receipt of a message, it can be given access to the internal definitions of the component.

Types and Values: TTCN-3 provides a set of basic and structured types, from which the user can derive own sub-types by restricting their values.

Template: a template is a data structure, that can be "used to either transmit a set of distinct values or to test whether a set of received values matches the template specification". When a template is used in the receiving direction to match with received values, each template can specify a set of values that it matches with.

Communication operations: TTCN-3 has both message- and procedure-based communication operations with which components can interact with each other and with the SUT.

Alternative behavior: in a test case, it is not always known beforehand in which order certain events occur. The SUT can have several legal actions it may perform, and it can behave completely erroneously. The situations in which several alternative events are possible are handled by TTCN-3 alt statement. The alt statement specifies a list of receiving operations (alternatives) The receiving operations are receive, getcall, getreply, catch, trigger, and check (explained in the previous section), with the addition of done and timeout. If the alternative matches with an event, then the code block following the alternative is executed, after which the execution continues after the alt statement, unless a repeat statement is encountered. If the alternative does not match, then all the following alternatives are tried in the order in which they are listed within the alt statement.

Altstep: altstep is a function like an element in TTCN-3 that can be used instead of the receiving operations in the alt statement.

Timers: TTCN-3 provides at language level syntax for specifying both implicit and explicit timers. The implicit timers are the timers whose values specify maximum execution time for test cases and function calls. These timers cannot or need to be started, read, or stopped by the user. Explicit timers are the user-created timers that can be started, read, and stopped, their timeout can be waited for, and they can be given as parameters to functions and altsteps. In the previous section, a timer was used in the context of the alt statement, to specify maximum time how long the component waits for messages to be received from the specified ports until it continues its execution.

Summary: TTCN-3 is a test specification language developed by ETSI that applies to a variety of application domains and levels of testing. TTCN-3 [26] was selected for this research study for its industrial strength to implement and execute TCs against SUT. It is designed for specifying collections of test cases in ATS that are then used to test the SUT. The top-level unit of TTCN-3 is a module that corresponds to a compilation unit in traditional programming languages. The module may contain a single case or several test cases that can be compiled or interpreted. A test case can be seen as the main function of a single case; it is always executed within an entity called a component to express its behaviour. The result of executing a test case is a verdict that determines if the SUT has passed the test. Listing 3-1 shows a test case that implements the DNS request introduced in the previous section.

```
1.  testcase VerifyDNSServer() runs on DNSClient {
2.   template String hostName := "MyHostName";
3.   template String IPAddress:= "192.124.135.56";
4.   clientPort.send(hostName);
5.   DNSTimer.start(10.0);
6.   alt {
7.     [] clientPort.receive(IPAddress) {
8.        setverdict (pass); }
9.     [] clientPort.receive {
10.       setverdict (fail); }
11.    [] DNSTimer.timeout {
12.       setverdict(fail) }
13.  }
14. }
```

Listing 3-1: TTCN-3 test case

A component should be defined (DNSClient) with a single port (clientPort) to communicate with the DNS server (SUT). The clientPort sends a data instance (*hostName*) to the SUT (line 4). Directly after, a timer is started (line 5) and set to run for 10 seconds. If the clientPort receives (line 7) the expected data instance (*IPAddress*) within 10 seconds, the test case passes. If the clientPort receives anything other than *IPAddress* (line 9) or the *DNSTimer* times out (line 11) the test case fails.

3.4 The Specification Level of the three Languages

The UCM language is used to describe the SUT behavior on requirement level (test goals), the resulting models abstract from detailed communication mechanisms and data which makes deriving executable test cases a difficult activity. On the other hand, TTCN-3 language is meant for specifying collections of test cases at the implementation level. The TTCN-3 test cases are developed and executed on the SUT when data becomes more subdivided and specific. This gap that exists between UCM models and TTCN-3 test cases can be filled by TDL language which allows describing tests on a higher abstraction level than a scripting language. Therefore, the TDL models can be used as an intermediate representation.

Each granular model of the three languages can be used to characterize a certain level of testing details. In particular, UCM models are developed at the requirement layer to abstractly formalize the functional requirement as test goals. The resulting test goals convey information to help develop some of the test specifications where test components and their interactions can be identified at the design layer. Finally, TTCN-3 test case implementation, developed at the test scripting layer, can be generated based on the obtained test specification. We claim that vertical transformation from the abstract test goals to a concrete test implementation can be achieved using the three languages. **Figure** 3.6 shows the link between the three languages and the models during model transformation activities.

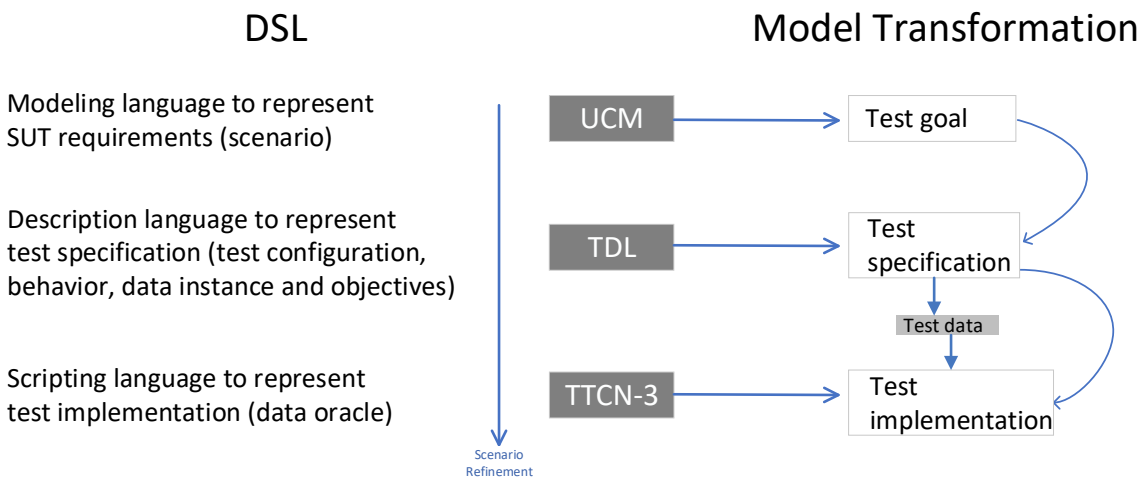


Figure 3.6: Link between the three languages and model transformation

3.5 Summary of Domain Specific Language

There are challenges in generating test cases from UCMs models as they reside at different abstraction levels from test cases. At first, tests need to be re-targetable and readable by test equipment, as supported by languages such as TTCN-3. Since UCM scenarios are abstract, there is a need to transform them to an intermediate level that help bridge the gap with the test cases in TTCN-3. In particular, the absence of elements such as alternative behavior and data in the UCM scenario metamodel makes generating test cases difficult as these elements are required for proper execution. Another challenge is the validation of generated test cases (TTCN-3) in terms of technical correctness, effectiveness and usefulness.

As a result, research is needed to explore and resolve the basic differences between UCM models and test cases in TTCN-3. Further research is also needed on when and how to introduce concrete data in the generation of executable test cases in TTCN-3, and how to link the generated artifacts among each other for traceability purposes.

Chapter 4 Towards Building a New Test Case Generation Approach

4.1 Research Questions

The conducted research and development study tackled the following problems: (1) difficulties in generating TTCN-3 test cases from abstract UCM models (2) delay in starting testing activities (3) substantial number of generated test cases to be checked, (4) weak links between requirement traceability with testing, and (5) high cost in achieving compliance with regulations and standard. The conducted research explored a model-driven testing paradigm to build a new testing methodology that covers two testing activities; (1) test case generation and (2) test case traceability.

The conducted research raised several questions that are centered on generating test cases and improving the testing process in terms of time and labor work:

Research Question 1: “how an existing legacy software tests can help in developing model transformation?”

Research Question 2: “what are some of the design factors a model transformation should have to bridge the abstraction gaps between UCM and TTCN-3 models to enable the generation of test cases?”

Research Question 3: “how do we assess the correctness of a test case generation process and how to evaluate its efficiency?”

Research Question 4: “how to align the activities of requirement traceability to testing to improve project cost and comply with DO-178C standards?”

The remainder of this chapter is organized as follows. The motivation to reengineer legacy software tests is first introduced in Section 4.2.1. The reengineering of legacy software tests activities to model-driven testing is presented in 4.2.2. This Section has two activities; the migration of legacy code to TTCN-3 code is presented in Section 4.2.2.1. Followed by code to

model activity presented in Section 4.2.2.2. Lesson learned from the reengineering activities is presented in Section 4.2.3. Section 4.2.4 concludes the chapter.

4.2 Reengineering Legacy Software Tests to MDT

The development of the test case generation process started by a modernization stage—reengineering the legacy software tests to model-driven testing. In particular the reengineering of the legacy test implementation to TTCN-3 and abstracting them to test specification in TDL models.

4.2.1 Motivation

At our research partner premises, the testing process (non-model based) to measure the quality of its prime product Flight Management System (FMS) is labor-intensive and error-prone. The FMS is a dynamical system i.e., system that evolves with time, a characteristic of such systems is the high dependency between events, the large amount of input and output data, making the test phase particularly challenging without the use of automation. The software test to verify the functionality of the FMS is developed manually from requirements. These requirements are expressed in NL and are layered as high-level requirements (HLR) and low-level requirements (LLR) in separate artifacts. The requirements are subsequently used as the basis, along with test engineer knowledge (implicit), for writing abstract test cases in NL, and then manually developing executable test cases using Eclipse Ant/XML software to test the SUT.

In this Chapter, we propose an approach, work published in a conference [129], that starts with the code migration of these legacy test cases to the TTCN-3 language, which in turn will be reverse-engineered into abstract TCs in TDL. Once the reengineering of the software tests is completed, new TCs can be captured directly in TDL, and these abstract TCs can be used to generate executable TCs in TTCN-3 or any other desired scripting language. Furthermore, when new requirements emerge to demand the evolution of the software tests, this software evolution can take place at the model level.

The ultimate goal in the reverse engineering process is to enable the automatic generation of the executable TCs and to have them migrate to a more standard testing language to benefit from its

important features. The next subsections explain the reengineering activities enclosed in the reverse engineering process, code-to-code migration and code-to-model.

4.2.2 Reengineering Activities

The reengineering of legacy software tests aims to discover feasible transformation from the test layer to test requirement layer, work presented in ETSI conference [130]. Furthermore, it is used to help build the model transformation, generate TTCN-3 test cases from TDL models, and show its feasibility. Then, after showing that TTCN-3 test cases can be derived from TDL models, the approach is extended with the requirement layer which describes software specifications in UCM scenarios where test objectives can be driven and transformed into TDL models. Reaching this point, the feasibility of transforming TTCN-3 scripts into a TDL model is shown, and a forward engineering process to regenerate the test cases can be undertaken.

Figure 4.1 shows two phases of the reverse-engineering process. The feasibility of transforming the legacy test cases into an abstract model is shown, along with a forward engineering process to regenerate the test cases in selected test language such as TTCN-3.

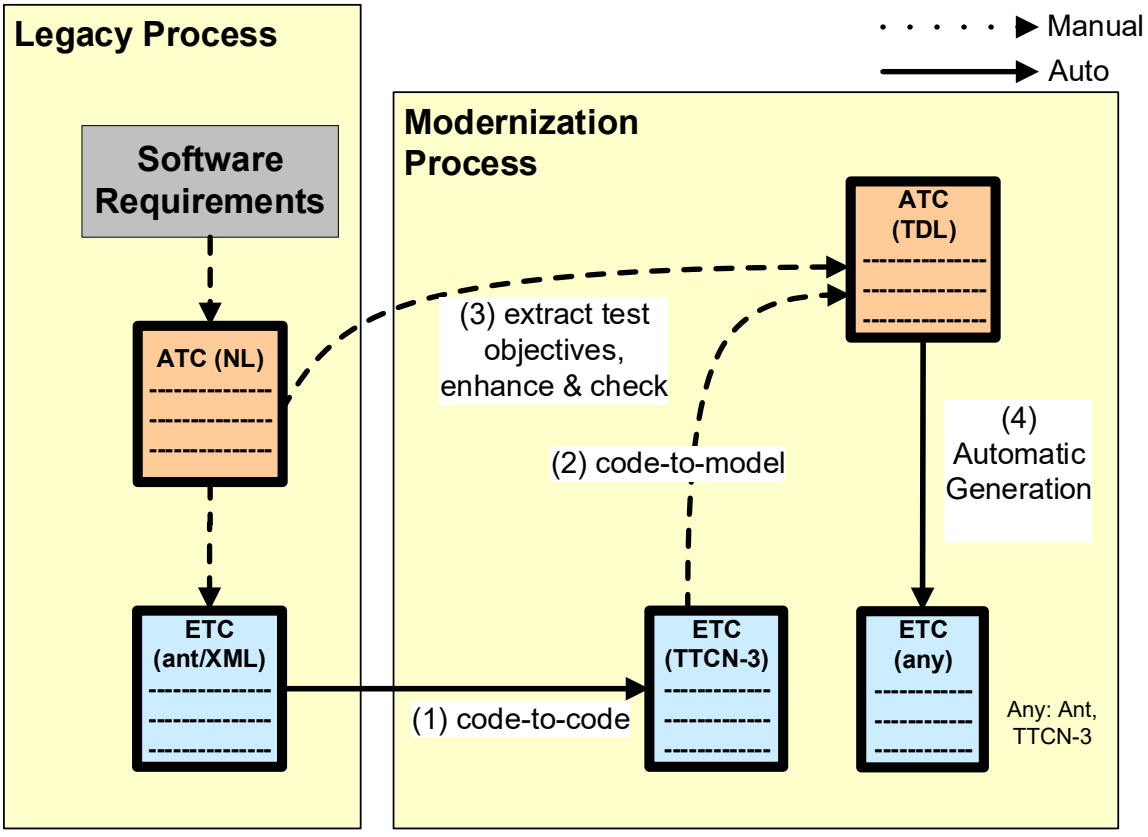


Figure 4.1: Modernization of legacy software tests

4.2.2.1 Code-to-Code Migration:

We developed a language translator tool to migrate the ETCs automatically to three TTCN-3 modules. This code migration is performed only once to obtain equivalent semantic code in TTCN-3. **Figure** 4.2 shows the architecture of the translator tool that generates three modules. The resulting modules along with a fourth module (Type module) constitute an executable TTCN-3 TC that is equivalent to the Ant/XML TC. The Type module is produced manually by analyzing the SUT inputs and outputs and the legacy. The architecture of the translator tool combines the following elements:

— Transformation Rules: several defined rules before the transformation of each Ant/XML construct to one or more equivalent constructs in TTCN-3. (one-to-many transformations are possible)

— Parser: reads legacy TC to generate syntactic element tokens encountered in the TP.

— Converter: based on transformation rules, it transforms the syntactic element, returned by the Parser, to functionally-equivalent code to the generator.

— Generator: writes the generated TTCN-3 code, produced by the Converter, dispatched in each of the corresponding modules.

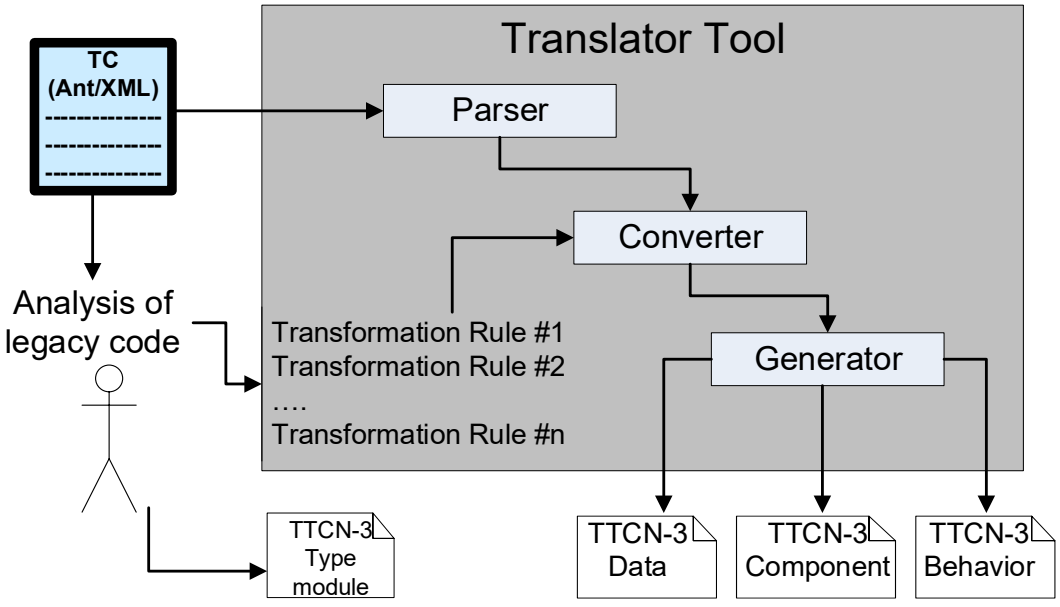


Figure 4.2: Language Translator Tool

Table 4-1 shows the transformation rules. The third column describes how the legacy ETC semantic is preserved using TTCN-3 syntax.

Table 4-1: Transformation rules to convert Ant/XML to TTCN-3 languages along with transformation rules

Legacy code element	Equivalent construct in TTCN-3	Transformation rules
<project name = "str">	module <str_Template> { } module <str_Behavior> { } module <str_Configuration> { }	Rule # 1: **project** element is translated to three **module** constructs which together compose a full TP in TTCN-3. The project name = str is used as a prefix with "Template", "Behavior" or "Configuration" to designate each TTCN-3 module. If the project name contains special characters such as dot or space, they are replaced by underscores.
<target name = "str">	testcase <target_str> runs on MTCType system SystemType	Rule # 2: **target** element is translated to a **testcase** construct, and the target name is prefixed with the string target_ The testcase will contain the action and verify constructs (stimulus and response)

<target name= "all" depends = "str_1, str_2, …, str_n"/>	control {
	execute (target_str_1());
	execute (target_str_2());
	execute (target_str_n());
	}
		Rule # 3: **target** name = all is translated to a **control** construct, and the intermediate targets, str_1, str_2, … separated by commas, identified in depends are translated to a sequence of **execute** statements such as execute (target_str_1()); in the **control** construct.

interface port = "name"	type port interface_name message {
	in sending_msg;
	out receiving_msg; }
	type component interfaceType {
	port interface *x* interface; }
		Rule # 4: Every interface is mapped to a message-based port and attached to a component.
		The **interface** port = name is translated to a type port message-based construct and attached to a type component construct.

<action key = "str_1", "str_2", …, str_n />	function action (name, command, str_1, …, str_n) runs on componentType {
	….
	portName.send(command, str_1, …, str_n);
	….
	}
		Rule # 5: **action** elements are translated to **functions** and **function** calls constructs. The action parameters command, name, str_1, …, str_n are passed as formal parameters to the function definition. The parameter name represents the interface name where command represents the input to send. Some actions take additional parameters to send the command, they can be represented by str_1, …, str_n. The parameter portName represents the port via which the input to SUT is sent. The action with its arguments in the legacy TP represent a stimulus to send to the SUT

< verify query = "str_1" value= "str_2" />	template component type verifyStep := {str_1 := **pattern** str_2 }
	function matchResult(verify, portName) runs on componentType {
	alt {
	[] portName.receive(verify) {
	setverdict(pass); }
	[] portName.receive {
	setverdict(fail); }
	[] replyTimer.timeout {
	setverdict(inconc, "No response from
	SUT") } }
		Rule # 6: **verification** is translated to **template** construct named verify. One **template** can host several **verifications** for a given step. Then, the construct verify is translated to a function to handle the alternative sequences. In the legacy TP, a comparison between the expected value and returned one is performed: verify query = "str_1" value= "str_2"
		The TTCN-3 TP migrates the expected values and store them in templates w.r.t to REGEXP used in the legacy. Then, the returned values are matched against the expected ones to issue a verdict.

< macrodef name = "MacroN" /> action <MacroN interface = "interface_name, $para_1$, $para_2$, …, $para_n$" />	function MacroN (interface_name, $para_1$, $para_2$, …, $para_n$) runs on componentType { …} MacroN(interface_name, $para_1$, $para_2$, …, $para_n$);	Rule # 7: **macros** elements are translated to **functions** and **function** calls constructs. The macros parameters interface_name, $para_1$, $para_2$, …, $para_n$ are passed as formal parameters to the function definition. A macro may contain control statement such as looping, if, else. These statements are mapped to their equivalent in TTCN-3

4.2.2.2 Code to Model

In the second phase of the reengineering process, we obtain the ATCs in TDL by reverse-engineering the migrated ETCs in TTCN-3. In most industrial domains, a test can be conceived at

two levels of abstraction: a test specification (or test case) and a test implementation (a test script). Our goal is to abstract the latter to obtain the former. Here, the test implementation is the migrated ETCs containing concrete information. It is often considered useful to express ETCs as stimulus-response scenarios. This is the path that we explore here using TDL.

Let's consider the modules of a ETC.

- The *Test Behavior* module is composed of test events (stimuli and responses as interactions) that express the test behavior.
- The *Test Data* module contains information about the test input and the expected test output.
- The *Test Component* module consists of a set of inter-connected test components with well-defined communication ports and an explicit test system interface.

An ATC should use abstract types and instances to refer to test data, and should describe the system components and their actions and interactions with a minimum of details. In our study, to raise the level of test specification, we choose the TDL notation. The TDL language was designed on three central concepts: (1) a Meta-Modeling principle that expresses its abstract syntax, (2) a user-defined Concrete Syntax for different application domains, and (3) the TDL semantics that can be associated to the meta-model elements. Any minimal TDL specification consists of the following major elements:

- A set of Test Objectives that specify the reason for designing either a Test Description or a particular behavior of a Test Description. It can be written as a simple text in NL and it can be complemented with tables and diagrams;
- A Test Configuration, which is a set of interacting components (tester and SUT) and their interconnection;
- A set of Data Instances used in the interactions between components in a test description.; and
- A set of Test Descriptions to describe one or more test scenarios based on the interactions of data exchanged between tester and SUT.

To obtain the ATC (TDL specification) from the ETC (TTCN-3 modules), we developed transformation rules to define ATC elements from the TTCN-3 ETCs'. These rules are meant for human processing; they are based on the equivalence between elements of both languages. The

rules aim to remodel the TTCN-3 modules into more abstract TDL elements. The language-sensitive editor understands the concrete TDL syntax, based on the TDL meta-model.

Next, we show how each TDL element is derived from its corresponding TTCN-3 module by applying these rules. However, extracting the TDL Test Objectives cannot be rule-based since the TTCN-3 ETCs do not have a concrete representation of the Test Objective. Nevertheless, the test objectives can be extracted from the legacy ATCs and copied in TDL corresponding elements.

I. Remodel Test Data Set

The concrete data definition, stored in the TTCN-3 *Test Data* module (TestData.ttcn3), is mapped to TDL Data Instances using TDL elements that link the data aspects between TDL and TTCN-3. These Data Instances are grouped in Data Sets and are considered as an abstract representation of the corresponding concepts in a concrete type system.

II. Remodel Test Configuration

In a TDL specification, the Test Configuration element consists of a Tester, SUT components and a Gate. The corresponding TTCN-3 *Component* module contains equivalent objects with many more details. Specifically, it consists of a set of interconnected test components with well-defined communication ports and an explicit test system interface. TDL does not have a *receive* construct, instead it uses a *send* construct for the interaction between a Tester and the SUT. Therefore, the mapping of TDL Tester and SUT components is validated with the TTCN-3 interaction.

III. Remodel Test Description

The Test Description element in the TDL specification language defines ATC behavior. The enclosed scenario is mainly composed of actions and interactions between the Tester and the SUT components.

In the TTCN-3 *Test Behavior* module, the action is a function implementation or physical setup. The interaction is represented as a message being sent (from a source) or received (from the target). We remodeled the interaction and the action to their equivalent in TDL by applying the rules listed in **Table** 4-2. In the *Test Behavior* module, numerous sequences of events are possible due to the reception and handling of communication timer events. The possible events are expressed as a set of alternative behaviors and denoted by the TTCN-3 *alt* statement. Each

TTCN-3 object in the *Test Behavior* is remodeled to an equivalent TDL construct by applying the transformation rules. In our experimentation, we used a TDL Editor to edit and validate the syntax of the TDL specifications.

Table 4-2: Transformation rules from TTCN-3 to TDL based on the proposed concrete syntax

TDL Meta-model elements (abstract syntax)	TTCN-3 statements	Our proposed TDL concrete syntax	Description of transformation from TTCN-3 to TDL
TestConfiguration	module <tc_name> { }	Test Configuration <tc_name>	Map to a Test Configuration statement with the name < td_name >
GateType	type port <port_type> message { }	Gate Type <port_type> accepts <Data_Set_name>	Map to a Gate Type statement with the name <port_type> that accepts Data Set elements
ComponentType	type component comp_type{ port <port_type> <port_name>; }	Component Type <comp_type> { gate types : <port_type> instantiate <comp_instance> as Tester of type <comp_type> having { gate <gate_name> of type <port_type> ; }	Map to a Component Type statement with the name <comp_type> and associate a <port_type> to it.
ComponentType	type component system_comp_type{ port <port_type> <port_name>; }	Component Type <comp_type> { gate types : <port_type> instantiate <system_comp_type> as SUT of type <comp_type> having { gate <gate_name> of type <port_type> ; }	Map to a Component Type statement with the name <system_comp_type> and associate a <port_type> as a port of the test system interface to it.
Connection	map (mtc: <comp_type>, system <system_comp_type>)	connect <comp_type> to <system_comp_type >	Map to a connect statement where a test component is connected to test system component.
TestDescription	module <td_name> { import from <dataproxy> all; import from <tc_name> all; }	Test Description(<dataproxy) <td_name> { use configuration: <tc_name>; { } }	Map to a Test Description statement with the name <td_name >. The <DataProxy> element passed as formal parameters (optional) is mapped from an import statement of the <DataProxy> to be used in the module. The import statement of the Test Configuration <tc_name> is mapped to use configuration property that is associated with the 'TestDescription'
Alternative Behaviour	alt {}	alternatively { }	Map to alternatively statement
Interaction	<comp_name_source>.send(<concreteData>)	<comp_name_source> sends instance < data_name > to <comp_name_target>	Map to a sends instance statement with respect to the sending component
	<comp_name_source>.receive(<concreteData>)	<system_comp_name_source> sends instance < data_name > to <comp_name_target>	Map to a sends instance statement when the sending source is SUT component
VerdictType	verdicttype <verdict_value>	Verdict <verdict_value>	Map <verdict_value> that contains the values: {inconclusive, pass, fail}to its corresponding value
TimeUnit	time_unit {1E-9,1E-6, 1E-3, 1E0, 6E1, 36E2	Time Unit <time_unit>	<time_unit> contains the following values: {tick,nanosecond,microsecond,miliisecond,second,minute,hour}
VerdictAssignment	setverdict (<verdict_value>)	set verdict to <verdict_value>	Map to a set verdict to statement
Action	function <action_name>()	perform action <action_name>	Map to perform action statement
Stop	stop	stop	Map to a stop statement within alternatively statement
Break	break	break	Map to a break statement within alternatively statement
TimerStart	<timer_name>.start(time_unit);	start <timer_name> for (time_unit)	Map to a start statement
TimerStop	<timer_name>.stop;	stop <timer_name>	Map to a stop statement
TimeOut	<timer_name>.timeout;	<timer_name> times out	Map to a times out statement
Quiescence/Wait	timer <timer_name> <timer_name>.start(time_unit); <timer_name>.timeout	is quite for (time_unit) waits for (time_unit)	Map to is quit for statement or to waits for
InterruptBehaviour	stop	interrupt	Map to interrupt statement

TDL Meta-model elements (abstract syntax)	TTCN-3 statements	Our proposed TDL concrete syntax	Description of transformation from TTCN-3 to TDL
BoundedLoop Behaviour	repeat	repeat <number> times	Map to repeat statement. The repeat is used as the last statement in the alternatively behavior.
DataInstance	var type <data_name>	Data Set <the_set> { instance <data_name> }	Map any <variable> to an instance and group it in Data Set element

This approach is suitable for automated ETCs as tests can be derived from the scenarios and automated.

4.2.3 Lesson Learned

There are some difficulties with the legacy process deployed, the test engineer spends a lot of time transforming LLR into executable test cases. There is a large gap in the abstraction level between the LLR and the executable test cases. The legacy scripts can be very large, difficult to maintain and hard to compose into complex scenarios involving parallelism. Their migration to TTCN-3 enforced coding standards and offered a more readable, simple to modify and easy to understand test code.

Formalizing LLR into TDL models for representing test descriptions allowed to validate easily the test requirements. Furthermore, as the detail level is low in the LLR, but very high in the scripts TDL models narrowed this gap by providing more formal details about the test interaction and configuration. The cost maintaining the migrated software tests becomes lower and less error-prone. In addition, TDL models are used both for communication between stakeholders and as the basis for implementing concrete tests.

Migration to a standards-based and more efficient software testing environment is appealing to organizations seeking to reduce costs, and to benefit from the continuing advancements in technology.

4.2.4 Conclusion

The modernization of software tests to a new platform is often pressured by business requirements to reduce the cost and effort of testing. In this study, we automatically restructured legacy test implementation, written as Ant/XML files into the TTCN-3 language that provides strong typing, structured constructs and modular code. Next, we reengineered the code and data

to a higher level of abstraction to obtain (model-driven) test implementation. Our overarching goal is to support test automation and to reduce the effort involved in testing.

The reverse engineering activities answered the research question RQ1: "*how an existing legacy software tests can help in developing model transformation?*"

Chapter 5 An MDTGL Approach for Testing Embedded Systems

5.1 Topic Overview

In this chapter, we proposed a new model-driven testing methodology, work published in the *Software & Systems Modeling* Journal [131], supported by a chain of tools that generates test cases to address an open problem about reducing test effort without forgoing the quality level of the final software.

Based on requirement propagation through model transformation, the new methodology aims to support the testing of embedded systems by generating TCs and maintaining requirement traceability. To do so, the approach relies on system models at different levels of abstraction. The primary contributions of this new testing methodology are:

i. The proposal of a new model-driven technique to generate TCs from abstract UCM scenarios at an early phase that is independent of any particular implementation of the design.

ii. The application of TCG approach during a feasibility study for the application of a functional testing process to industrial avionics applications.

iii. The validation of the test case generation approach in comparison with the industrial testing process.

iv. The proposal of a new framework to strongly link the activities of requirement traceability with generated test cases.

To validate the efficiency of the new methodology in terms of generating TCs and correct workflow, we applied it to a real case study in the aviation industry. The validation and comparison process are based on analyzing the generated test artifacts by performing requirement-based test coverage and verdict analysis. We used a case study approach to address the raised questions. Two case studies from the avionics domain were used to build the new testing methodology, collect the data, demonstrate the feasibility, and assess the effectiveness.

The remainder of this chapter is organized as follows. The research methodology used to solve the problems is presented in Section 5.2. The proposal of a model-driven testing methodology is presented in Section 5.3. The first testing activity of the MTDGL is the generation of test cases which is explained and demonstrated in great detail in Section 5.3.1. Followed by the traceability links activity in Section 5.3.2. Section 5.4 concludes the chapter.

5.2 The Research Methodology

The research study was conducted at our research partner premises who is a world leader in the design and manufacture of high-technology electronics products for aviation. At Avionic industry labs, the testing process (non model-based) to measure the quality of its prime embedded system FMS is labor-intensive and error-prone.

Our research study used a case study method to tackle the problems and build an automated new approach. We used industrial case studies for demonstrating the approach applicability and assessing its effectiveness.

5.2.1 Conducted Research

The conducted research covered the following:

- Reversed-engineer of legacy software tests that validate the FMS software to be driven from models.
- Built a test case generation approach that is composed of independent layers;
    i. Requirement layer (Abstractly formalized functional requirements)
    ii. Test design layer (Identified test components and their interactions)
    iii. Test scripting layer (Generated test cases)
- Enabled information transformation between the first three layers (i→ii→iii) by using concepts such as abstraction, model transformation, and successive refinement.
- Developed a traceability framework to record traceability links among the generated testing artifacts.
- Applied the new approach to safety-critical software such as LGS to assess its feasibility; layers (i, ii, and iii)
- Assessed the effectiveness of the approach by applying it to real case study FMS and compared the obtained workflow to the legacy one; all layers.

5.2.2 Collected Data

— Collected data (functional requirement) from LGS case study and use it as a running example to demonstrate the applicability of the proposed approach. The LGS is a public case study from the avionics domain.

— Collected data (functional requirements in NL, legacy executable software tests are written Ant/XML and test results that store the execution traces of the FMS with its various interfaces) from FMS case study and use it to analyze and assess the effectiveness of the proposed approach. The FMS is a real case study from the avionics domain developed at our research partner premises and used as legacy software to test the FMS implementation.

5.2.3 Facilities Used

The facilities used for the research are the following:

Software:

— jUCMNav – A modeling tool: jUCMNav is a free, Eclipse-based graphical editor and an analysis and transformation tool for the User Requirements Notation (URN).

— TDL Editor – A test editor tool: TDL Editor is a private tool to edit, design, document, and represent formal test descriptions. The Editor defines the specific domain of the TDL language and is based on its meta-model.

— TTworkbench – A test script editor tool: TTworkbench is a full-featured integrated test development and execution environment (IDE). This tool allows testing of software products and services. The tool supports the TTCN-3 ETSI standard. (An academic license is obtained from Spirent Company).

— Xtext – A framework for the development of programming languages and DSL.

— Xtend – Is a general-purpose high-level programming language used for generating code.

— Eclipse – Eclipse is an integrated development environment (IDE) for developing Java applications

— Eclipse Modeling Framework (EMF) – The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model.

— Simulation of FMS Application – An FMS/PTT is a simulation of FMS product developed by our research partner (A copy of the application is obtained)

Hardware:

— A personal computer with Windows operating system.

— Dongle Key to run the FMS simulation.

5.3 The Methodology MDTGL

This section presents the new methodology MDTGL for testing embedded system, the methodology includes two major testing activities; (1) generating TCs and (2) maintaining traceability links among the generated testing artifacts.

5.3.1 Test Case Generation Approach

The test case generation (TCG) approach shown in **Figure** 5.1 starts when the test designer wants to describe the NL requirements into behavioral models. This activity answers the research question RQ2: **"***what are some of the design factors a model transformation should have to bridge the abstraction gaps between UCM and TTCN-3 models in order to enable the generation of test cases?***"**

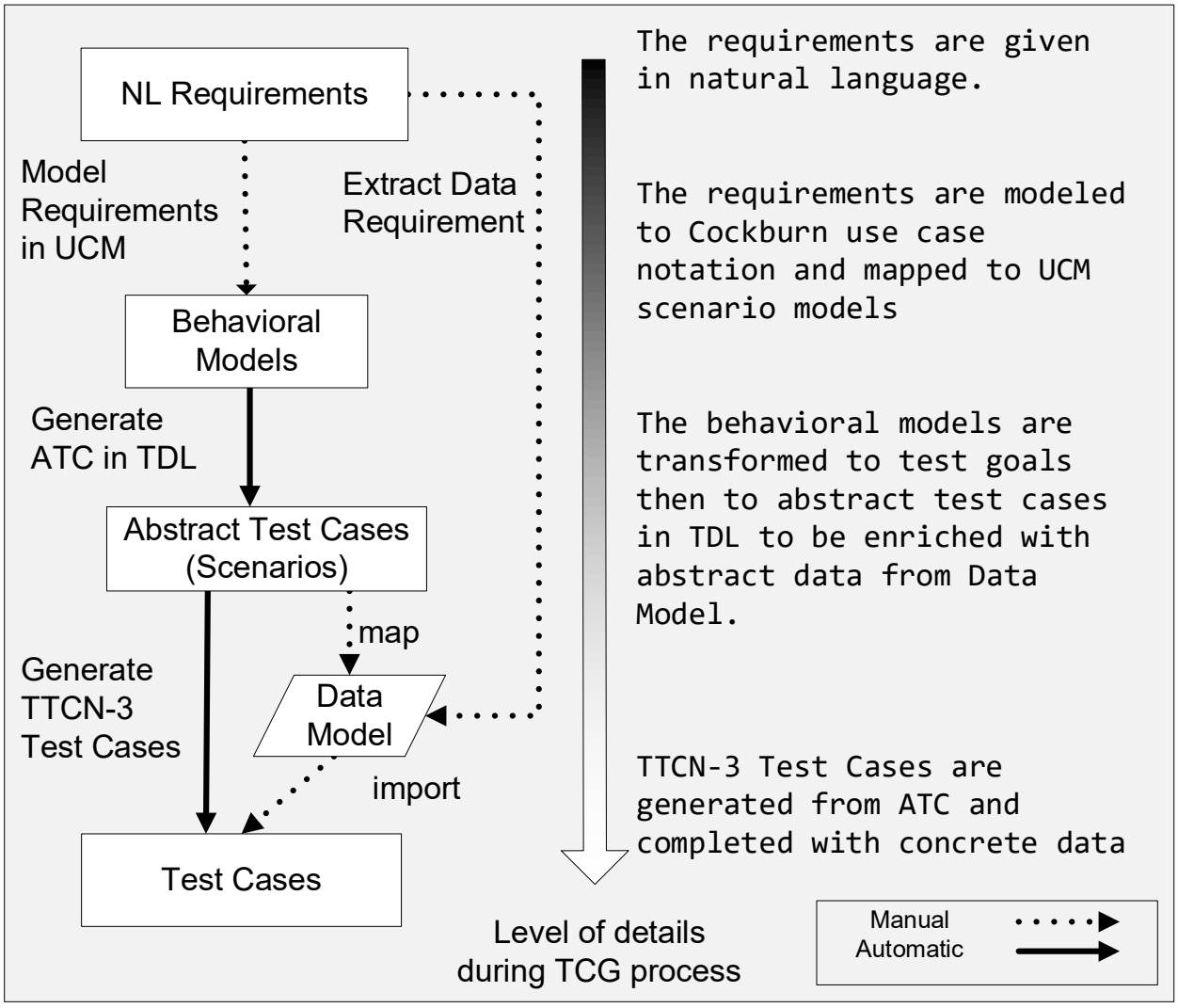


Figure 5.1: TCG approach for testing an embedded system

The key points of the TCG approach are: (1) NL requirements are described in behavioral models; (2) These models are exported to test goals and transformed, based on developed rules, to ATC that are completed manually with data instances; and (3) the obtained ATCs are transformed, based on developed rules, along with concrete test data to TCs.

The approach can be seen as a process of successive refinements of specifications that involves model transformation and the insertion of additional information. The approach must ensure test effectiveness— all requirements are covered— while also aiming for test efficiency— the testing effort is reduced by decreasing the manual development while ensuring the discovery of

implementation errors in the SUT. The approach offers features that should be attractive to test designers, such as scenario coverage and a simple structure, where ease of use and understandability are key.

In the following subsections, we explain how requirement propagation through model transformation and insertion of additional information are performed at each step in the process. A case study is conducted in Section 5.3.1.5 to demonstrate the feasibility of the approach.

5.3.1.1 Formalizing SUT Requirements into Behavioral Model

In order to facilitate the modeling of the NL requirements into UCM elements, the requirements are written in Cockburn use case notation [132]. With some basic knowledge of the jUCMNav tool, the modeled use case is mapped manually to UCM scenarios models.

The scenario models represent the system from a functional execution sequence perspective, which is another form in which to represent the system and software requirements. Scenarios provide benefits for system comprehension, design, testing and maintenance. Scenarios can be grouped, related and decomposed for better management, reusability and analysis. Furthermore, scenarios can be used later in the verification process to drive the test specification and to direct the development of TCs.

In our TCG approach, UCMs are an intermediate step towards deriving abstract test descriptions.

5.3.1.2 Transform Behavioral Model into ATC

A UCM scenario model conveys information to help develop some of the TDL specification elements, in particular, *Test Objective*, *Test Configuration,* and *Test Description*.
Since UCM scenarios deal only with behaviour, the concept of data is yet to be supported. Therefore, we developed a data metamodel, see **Figure 5.2**, that is based on test data requirements to help identify UCM *responsibilities* that exchange messages, develop the TDL *Data Sets*, and detail the TTCN-3 data with concrete values.

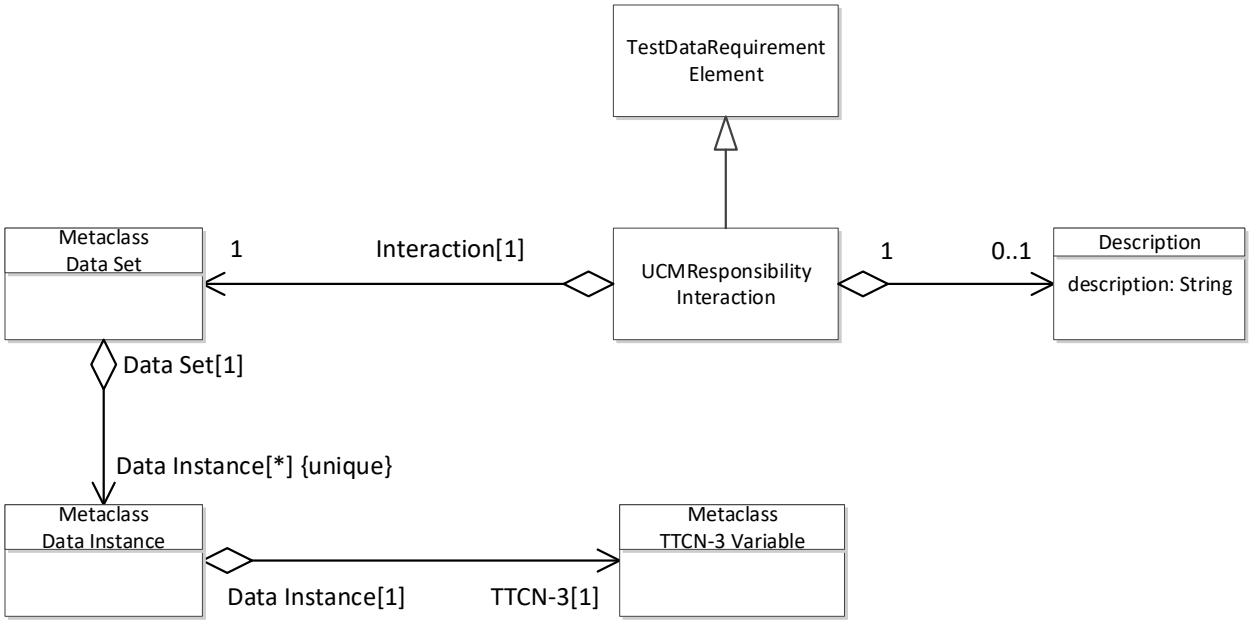


Figure 5.2: Data metamodel

Next, we developed a process called *ATC Builder* as shown in **Figure** 5.3, to transform the UCM scenario model and data model (additional information) into an ATC expressed as a valid TDL test specification.

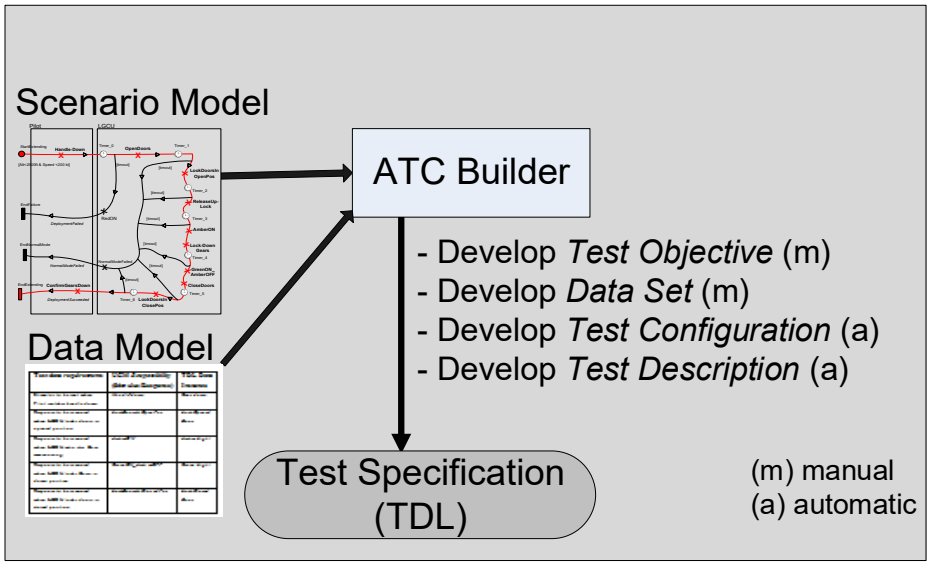


Figure 5.3: ATC builder process

The *ATC Builder* process transforms the UCM scenario to four TDL elements. The development of each element is shown in the following:

I. Develop Data Set

In general, the test inputs for the TCs are produced in the test analysis and design process. We assume that it is possible to select enough data from the analysis process to enable the development of test input for use in the TCs.

A *responsibility* definition in UCM scenario represents an action or the steps to perform, either informally through its name or more formally with the help of its expression. Using this information, the *responsibilities* involved in a stimulus/response action is flagged as interaction messages and mapped into *Data Instances* in TDL. A data model based on data requirements composed of three levels of test data abstraction is developed:

a) **Stimulus/Response:** a subset of test data requirements can be represented abstractly as I/O message in UCM *responsibility* objects;
b) **Test data scenario:** the I/O messages in the Stimulus/Response level are developed into a TDL *Data Sets*.
c) **Test data procedure:** The *Data Sets* are developed using templates.

Table 5-1 shows four columns of test data: the test data requirements, the complete set of UCM *responsibilities,* and its corresponding TDL *Data Instances* and TTCN-3 *Data Templates*.

Table 5-1: Test Data for UCM scenario

Test data requirements	**UCM *Responsibility* (Stimulus/Response)**	**TDL Data Instance**	**TTCN-3 Template**
Stimulus/Response to be exchanged	*Interaction*	*Data Instance*	*Data Template*

Each UCM responsibility in the second column (interaction) is either a stimulus to send or a response to receive. This interaction is represented as a TDL *Data Instance* in the third column and as a TTCN-3 *Data Template* in the last column.

The *Data Instances* to be used in the *Test Description* are developed manually and grouped in *Data Sets.* They are an abstract representation of the corresponding data-related concepts in a concrete type system.

II. Develop Test Configuration and Test Description

A *Test Configuration* in TDL specifies the communication infrastructure necessary to build upon the *Test Description*. As such, it contains all the elements required for the exchange of

information, such as *Component Instances* and *Connections*. Each *Component Instance* specifies a functional entity of the test system. A *Component Instance* may either be a part of a Tester, or a part of an SUT. The *Test Configuration* element consists of:

- A Tester;
- SUT components;
- A Gate[1]; and
- Interconnections between Tester and SUT components via Gate instance.

The metamodel of *Test Configuration* and *Test Description* are shown in Appendix B and Appendix C respectively.

The *Test Description* element defines the expected behaviour, the actions and the interactions between system components. The *Test Description* is associated with exactly one *Test Configuration*, and may be associated with any number of data elements that represent the formal parameters. Any number of *Test Objectives* can be attached manually to the *Test Description* to help to specify its design.

The *Test Description* in TDL defines the test behaviour based on ordered atomic or compound behaviour elements. A *responsibility* object in a UCM scenario model represents an action to be performed by its enclosing component. Its equivalence in TDL is mapped to *Action Reference* element, which is an atomic behaviour used to refer to an *Action* element to be executed. The dynamic and static *stub* objects that contain sub-maps are not mapped to any TDL element, but their enclosed *responsibilities* are. An *Action Reference* may have a *Component Instance* attribute identifying the component instance on which the action is to be performed. Any information exchanged via the gates is represented abstractly, and can be referenced by TDL *Interaction* elements. An interaction can represent a message sent from a source and received by a target.

In our approach, we used the feature path traversal algorithm in the jUCMNav tool to export UCM scenario models in XMI format. We developed a java tool to parse the exported scenario and transform it automatically to TDL *Test Configuration and Test Description* elements. The exported scenarios are structured by a metamodel, see **Figure** 5.4, and as such can be handled by

[1] A Gate is a point of communication for exchanging information between components, it specifies also the data that can be exchanged

the model transformation. The exported scenarios have exhaustive coverage of the UCM model. The algorithm uses a depth-first traversal [133] of the scenario that captures the UCMs' structure.

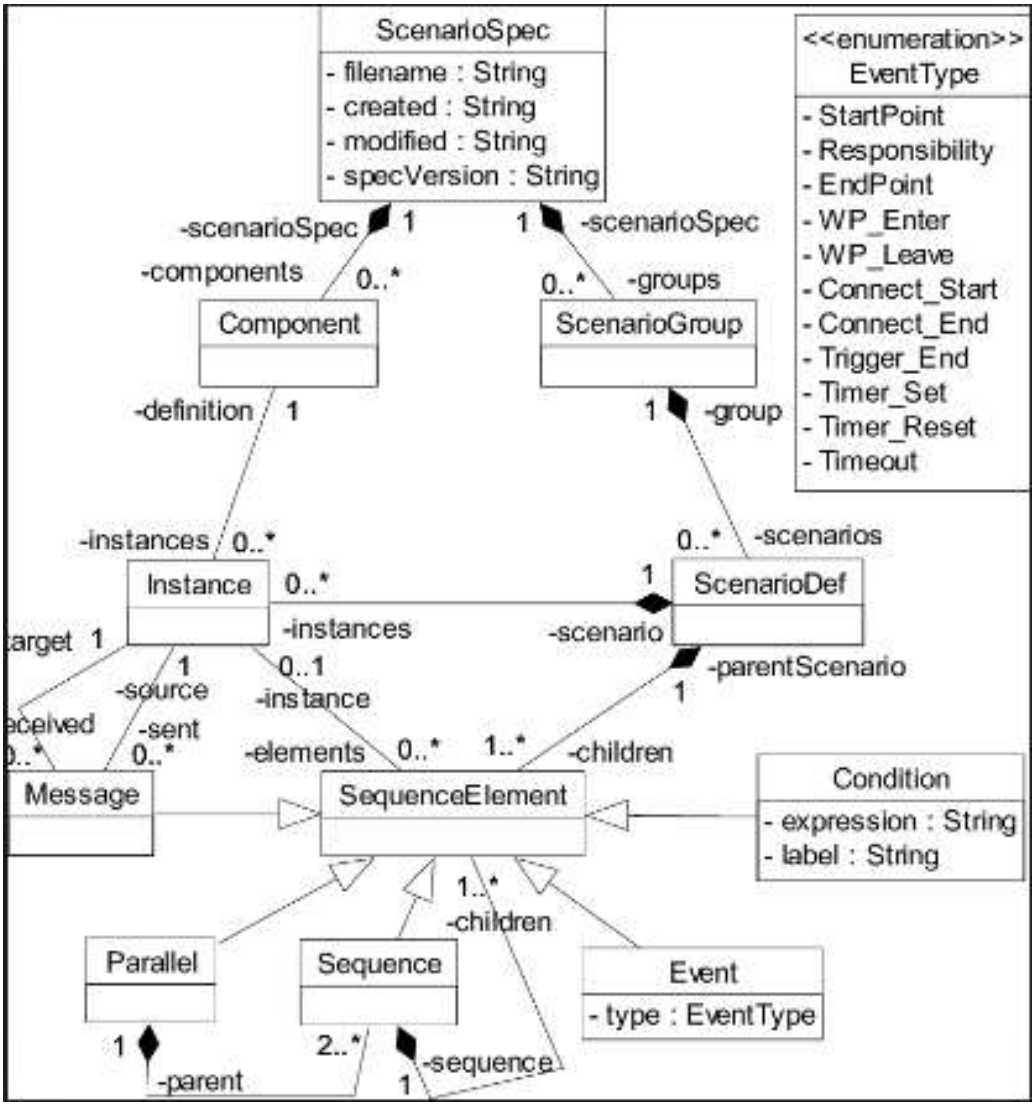


Figure 5.4: Scenario definition Metamodel

The algorithm traverses the path elements beginning at a start point until a stop point (AND-join, waiting place, or timer) is reached. Then, the algorithm backtracks to get the next available branch of an AND-fork (unvisited branches) or the next start points if any. The traversal is successful if all elements along the path are marked as visited. The algorithm can prevent infinite loops through a maximum number of visits. The exported scenario contains traversed UCM elements such as *Packaged Element, Component Instance, Gate Instance, Action Reference, Interaction, etc*. that

we use to develop the TDL Specification that can be compiled in the proposed TDL concrete syntax.

The java tool parses the exported scenario using XMLStreamReader interface and automatically generates the two TDL elements *Test Description* and *Test Configuration.*

The interface XMLStreamReader is used to iterate over the various events in the exported scenario to extract the information and convert it to TDL syntax. Once we are done with the current event, we move to the next one and continue till the end of the scenario. The events can be for example the start of an element, the end of element or attribute. **Figure 5.5** and **Figure** 5.6 illustrates the development of the TDL *Test Configuration* and *Test Description* from UCM scenario. Our tool iterates over the TDL elements represented with abstract syntax in the exported scenario and transforms it to concrete syntax in TDL notation.

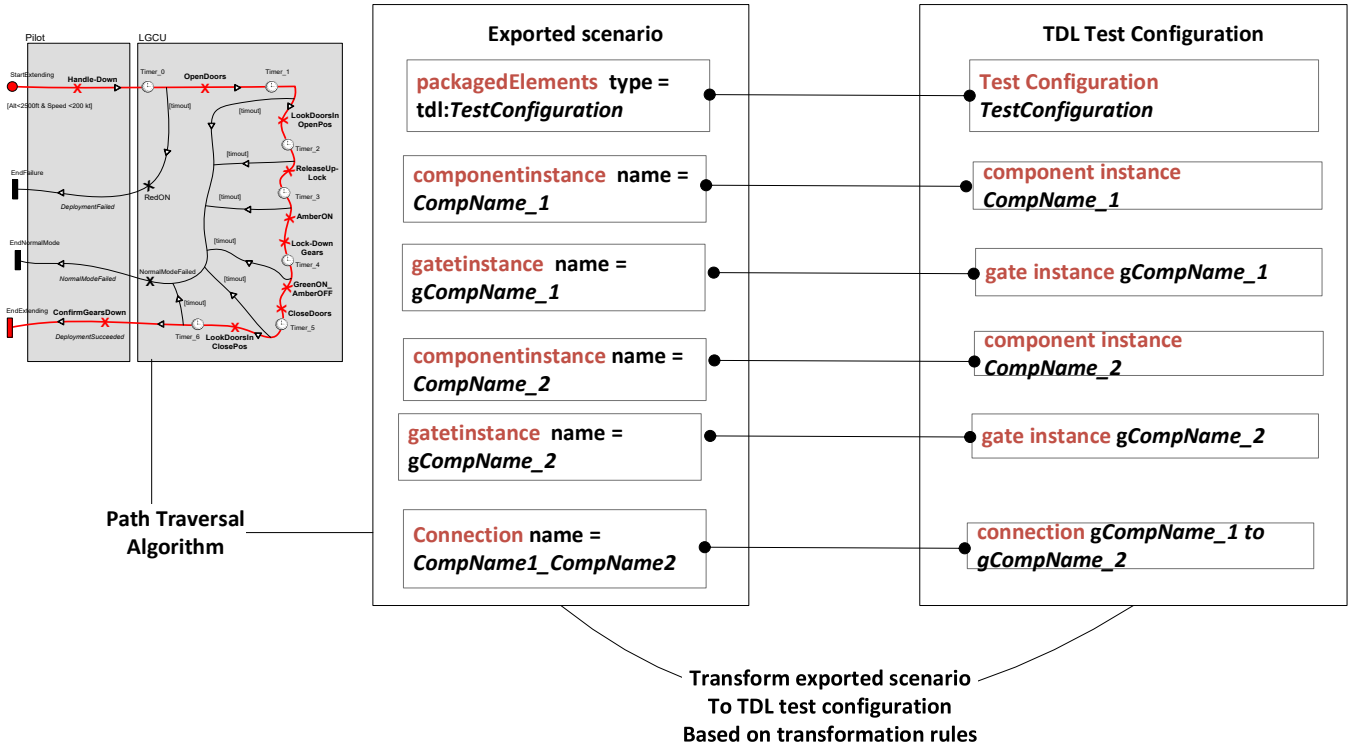


Figure 5.5: The development of TDL Test Configuration

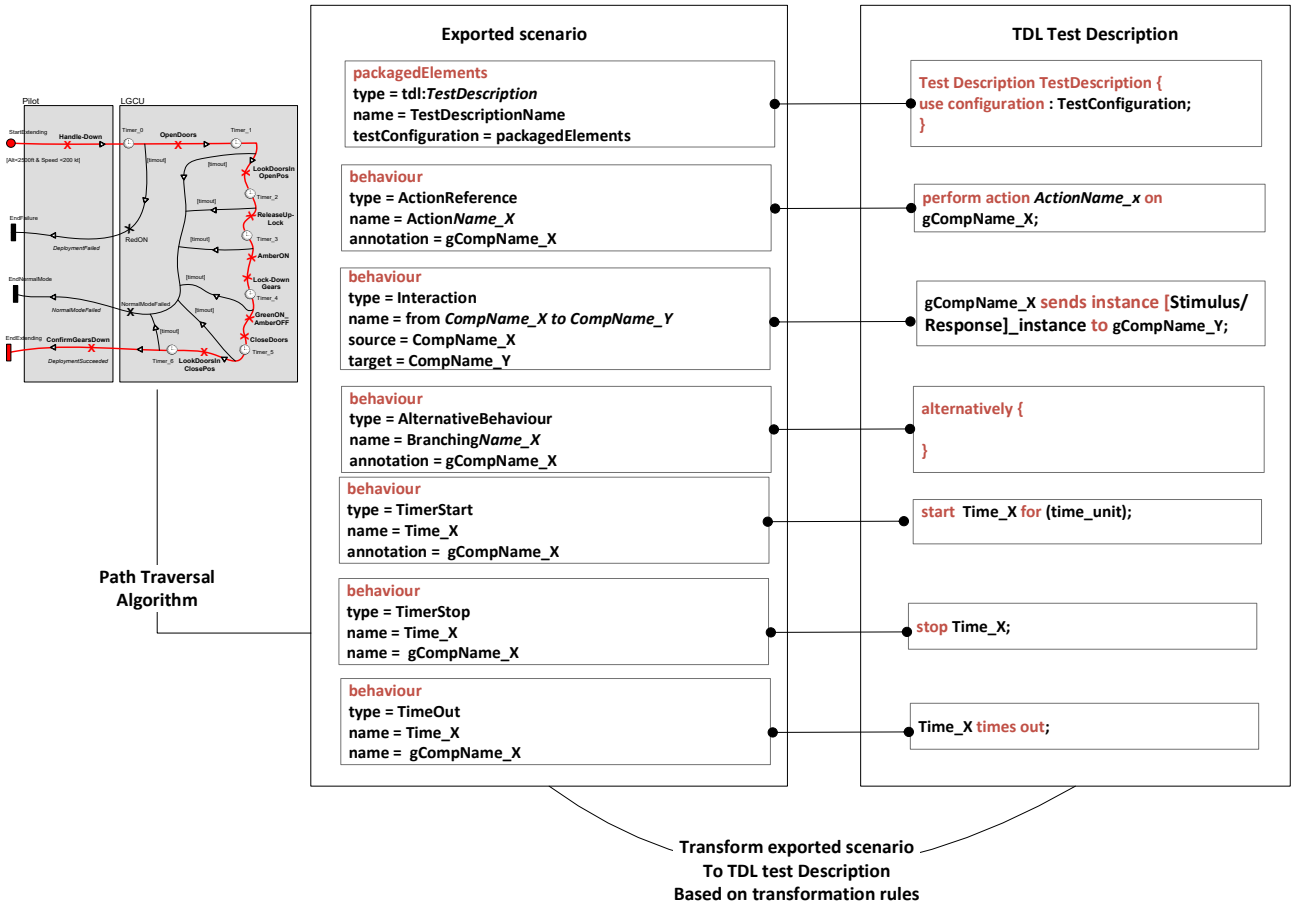


Figure 5.6: The development of TDL Test Description

III. Develop Test Objective

TDL *Test Objectives* are developed by analyzing the exported scenario definition. *Test Objectives* set guidelines to design the *Test Description* or to design a particular behaviour. Typical UCM objects include *component, responsibility, comment, timer,* and *condition*. The *Test Objective* can be enriched by adding additional information from the system requirements.

IV. Post-Processing of Alternative Behavior

The transformation algorithm from behavioral model to ATCs generates only linear scenarios or one alternative per scenario while a typical ATC in TDL has alternative responses. Therefore, it requires at UCM level either a different traversal mechanism with a different scenario metamodel, or post-processing of scenarios to merge those that constitute alternate test behaviors. In our approach, we automated the post-processing of alternative behavior. The technique developed selects the common interaction behavior that represents different responses to the tester and groups them in the alternative element as illustrated in **Figure** 5.7.

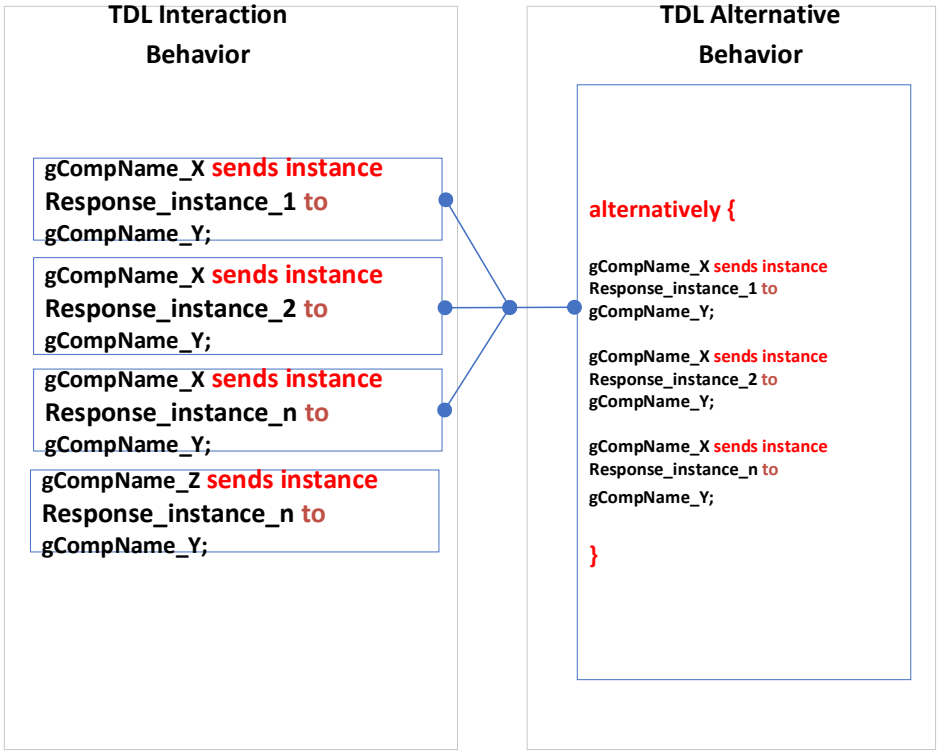


Figure 5.7: Post-processing of alternative behavior

Finally, the resulting elements are combined along with *Test Objectives* and *Data Sets* in one TDL Specification and used a TDL Editor[2] to edit and validate the specification. The Editor defines the specific domain of the TDL language and is based on its metamodel. The DSL of the TDL is written in the *Xtext* language development framework [134].

We made the java tool and TDL Editor available online[3]; interested readers can download the eclipse project to generate TDL *Test Configuration* and *Test Description* elements from UCM scenarios. The TDL Specification is based on the TDL meta-model and expressed in concrete syntax. It clearly separates the ATC from its associated TC by providing an abstraction level. As a result, the test designer can focus on describing an ATC that covers the given *Test Objectives* rather than fully implementing the script. It is the final implementation as a TC that will ensure the discovery of implementation errors in the SUT.

[2] Obtained from Philip Makedonski, University of Göttingen.

[3] https://users.encs.concordia.ca/~bentahar/Model_Transformation/

5.3.1.3 Transform ATCs into TCs

The derivation of test specifications from a UCM scenario model is an abstraction of the expected behavior between components and cannot be used directly on the actual SUT. The ATCs thus described lack concrete details about the SUT and its environment. Therefore, TCs should be derived and sufficiently detailed with test data and interface requirements (additional information) to correctly communicate with the SUT. We propose to use TTCN-3 language to implement the ATCs defined by the TDL specification package. The document TTCN-3 Core Language [135] defines the syntax of TTCN-3 using extended BNF.

One of the design objectives of TDL is to be less technical and thus user-friendly for non-technical users and that it can serve as the basis for the implementation of executable tests that are by definition highly technical.
Based on transformation rules that we developed between TDL source and TTCN-3 target, the ATCs are transformed into TCs. The technique that we applied in this model transformation is structural, e.g., a TDL element, shown in italics, is transformed into a TTCN-3 module. Therefore, we consider that an executable test suite in TTCN-3 is broken down into four types of modules: (1) a Test Configuration module that consists of a set of inter-connected test components with well-defined communication ports, (2) a Test Description module which usually contains behavioral program statements that specify the dynamic behavior of the test components over the communication ports, (3) a Test Oracle module that contains templates (expected result or responses) used to test whether a set of received values matches the template specifications, and (4) Test Input module that contains input data (stimulus) to be transmitted to the SUT. This modular approach of deriving the TCs supports the model transformation between source and target elements and promotes the reusability of the generated modules. **Figure 5.8** shows the derivation of the executable test suite in TTCN-3.

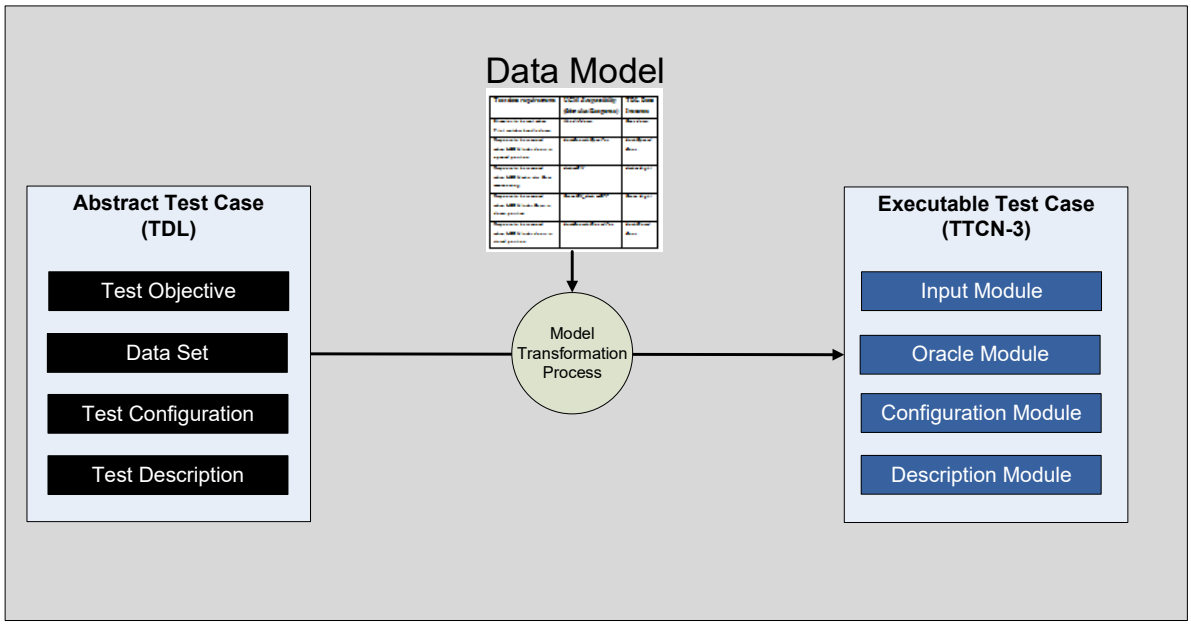


Figure 5.8: Derivation of ETC in TTCN-3

The transformation rules that enable the transformation of TDL specification, listed in **Table** 5-2, are programmed and implemented in a tool based on model-to-text technology called *Xtend.* We made the tool available online[4].

Table 5-2: Transformation rules from TDL model to TTCN-3 constructs

	TDL Meta-model elements (abstract syntax)	Our TDL concrete syntax	Equivalent TTCN-3 statements	Description
Rule# 1	TestConfiguration	Test Configuration <tc_name>	module <tc_name> { }	Map to a module statement with the name < td_name >
Rule# 2	GateType	Gate Type <gt_name> accepts dataOut, dataIn;	type port <gt_name> message { inout dataOut; inout dataIn; }	Map to a port-type statement (message-based) that declares concrete data to be exchanged over the port.
Rule# 3	ComponentType	Component Type <ct_name> { gate types : <gt_name> instantiate <comp_name1> as Tester of type <ct_name> having { gate <g_name1> of type <gt_name> ; }	type component comp_name1{ port <gt_name> <g_name1>; }	Map to a component-type statement and associate a port to it. The port is not a system port.

[4] https://users.encs.concordia.ca/~bentahar/Model_Transformation/

Rule# 4	ComponentType	Component Type <ct_name> { gate types : <gt_name> instantiate <comp_name2> as SUT of type <ct_name> having { gate <g_name2> of type <gt_name> ; }	type component comp_name2{ port <gt_name> <g_name2>; }	Map to a component-type statement and associate a port of the test system interface to it.
Rule# 5	Connection	connect <g_name1> to <g_name 2>	map (mtc: <g_name1>, system: <g_name2>)	Map to a map statement where a test component port is mapped to a test-system interface port
Rule# 6	TestDescription	Test Description(<dataproxy) <td_name> { use configuration: <tc_name>; { } }	module <td_name> { import from <dataproxy> all; import from <tc_name> all; testcase _TC() runs on comp_name1 {} }	Map to a module statement with the name <td_name >. The TDL <DataProxy> element passed as a formal parameter (optional) is mapped to an import statement of the <DataProxy> to be used in the module. The TDL property test configuration associated with the 'TestDescription' is mapped to an import statement of the Test Configuration module. A test case definition is added.
Rule# 7	AlternativeBehaviour	alternatively { }	alt {}	Map to an alt statement
Rule# 8	Interaction	<comp_name1> sends instance <instance_outX> to <comp_name2>	<comp_name1> .send(<instance_outX>)	Map to a send statement that sends a stimulus message
		<comp_name2> sends instance <instance_Inx> to <comp_name2>	<comp_name1> .receive(<instance_InX>)	Map to a receive statement that receives a response when the sending source is an SUT component.
Rule# 9	VerdictType	Verdict <verdict_value>	verdicttype	<verdict_value> contains the following values: {inconclusive, pass, fail}. No mapping is necessary since these values exist in TTCN-3
Rule# 10	TimeUnit	Time Unit <time_unit>	N/A	<time_unit> contains the following values: {tick, nanosecond, microsecond, millisecond, second, minute, hour}. No mapping is necessary; a float value is used to represent the time in seconds

Rule# 11	VerdictAssignment	set verdict to <verdict_value>	setverdict (<verdict_value>)	Map to a setverdict statement.
Rule# 12	Action	perform action <action_name>	function <action_name>() runs on <g_name1>{ } <action_name (); >	Map to a function signature and to a function call. The function body is refined later if applicable.
Rule# 13	Stop	stop	stop	Map to a stop statement within an alt statement.
Rule# 14	Break	break	break	Map to a break statement within an alt statement.
Rule# 15	Timer	timer <timer_name>	timer<timer_name>	Map to a timer definition statement.
Rule# 16	TimerStart	start <timer_name> for (time_unit)	<timer_name>.start(time_unit);	Map to a start statement.
Rule# 17	TimerStop	stop <timer_name>	<timer_name>.stop;	Map to a stop statemen.t
Rule# 18	TimeOut	<timer_name> times out	<timer_name>.timeout;	Map to a timeout statement.
Rule# 19	Quiescence/ Wait	is quite for (time_unit) waits for (time_unit)	timer <timer_name> <timer_name>.start(time_unit); <timer_name>.timeout	Map to a timer definition statement, a start statement and to a timeout statement.
Rule# 20	InterruptBehaviour	interrupt	stop	Map to stop statement
Rule# 21	BoundedLoopBehaviour	repeat <number> times	repeat	Map to a repeat statement. The repeat is used as the last statement in the alt behaviour. It should be used once for each possible alternative.
Rule# 22	DataSet	Data Set <DataSet_name> { }	type record <DataSet_nameType> { } template <T_DataSet_nameType> := { }	Map Data Set to record type and template using DataSet_name, T_ DataSet_name and prefixed with “Type”
Rule# 23	DataInstance	instance <instance_name>;	[<instance_name_S>;] [<instance_name_R>;]	Map instance to a variable, using instance_name and prefixed either with “_S” for stimulus or with “_R” for response

In our approach, the TDL elements developed previously were used, based on transformation rules, to derive the corresponding modules in TTCN-3. Next, the derived Test Input and Test Oracle modules were enriched with concrete data from the data model to enable the execution of the TCs. The development of the TTCN-3 modules is discussed in the following:

I. Generate the Input and Oracle Modules

As mentioned earlier, TDL does not offer a complete data type system. Instead, it depends on *Data Set* elements— whose *Data Instances* are an abstract representation of the corresponding data-related concepts in a concrete-type system. Therefore, the *Data Instances* developed in the previous section can be used along with data requirement analysis to develop concrete data definition. In our approach, a TTCN-3 data module that contains test input and test oracle definitions based on *Data Sets* is developed. The language *Xtend*, part of the Eclipse *Xtext* project, is used to generate partial TTCN-3 code from TDL *Data Set* syntax. All *Data Set*

instances can be identified from the parsed TDL model and generate a record type and a template for each in TTCN-3 syntax, see **Figure 5.9**.

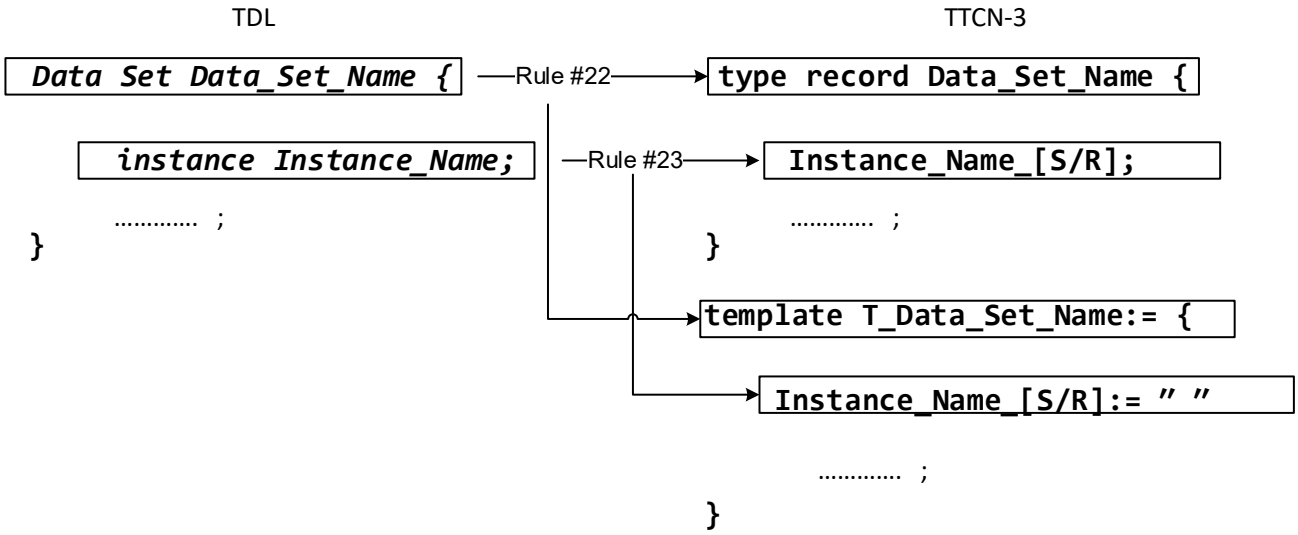


Figure 5.9: TDL Data Set transformation

After the TTCN-3 data module is partially generated and test data becomes available, the module is completed with test oracle information and typed with concrete TTCN-3 types.

II. Generate the Configuration Module

When describing a *Test Configuration* in TDL, the main focus is usually on the test components and their communication, whereas an executable test requires a more detailed configuration. A *Test Configuration* in TDL consists of Tester and SUT components, gates, and their interconnections represented as the *Connection*. A TTCN-3 configuration should consist of a set of inter-connected test components with well-defined communication ports and an explicit test system interface which defines the boundary of the test system. Furthermore, the communication between components is achieved via well-defined port types such as message-based and procedure-based ports. The transformation rules in **Table** 5-2 are used to enable the transformation of an ATC to a concrete TCs. The TDL *Test Configuration* contains the necessary objects, test components and communication channels to build the TTCN-3 configuration module. The concrete details needed to communicate correctly with the SUT, such as the message type to be sent or received, are imported from the TTCN-3 data module where the test inputs and test oracle are defined. The TDL *Test Configuration* components such as gate, Tester,

and SUT are transformed to equivalent objects in TTCN-3 by applying Rule #2, #3 and #4 as shown in **Figure 5.10**.

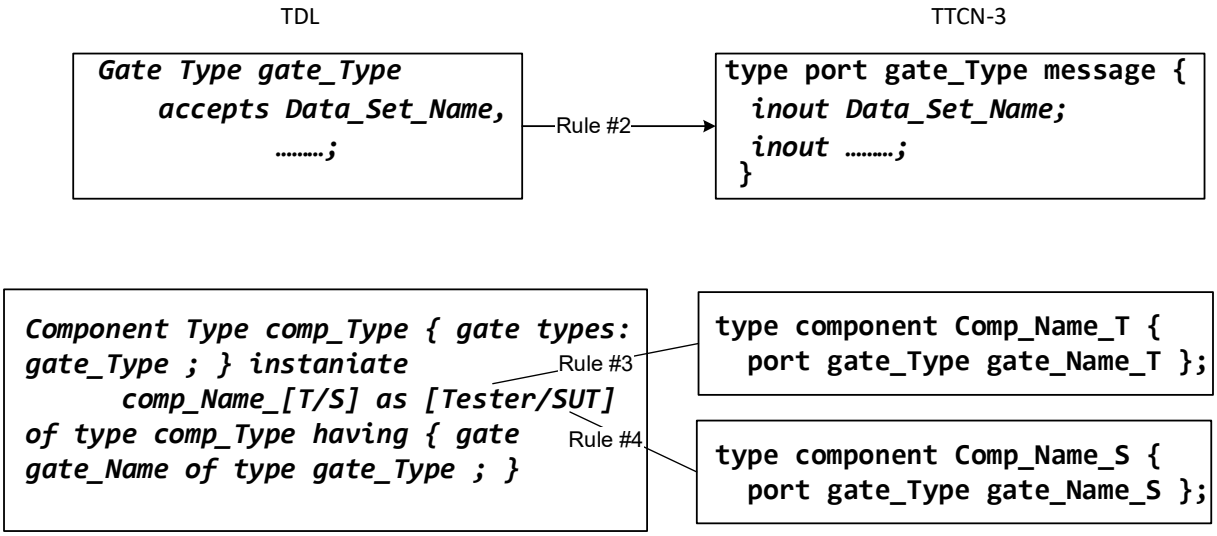


Figure 5.10: TDL Test Configuration transformation

More specifically, these rules are implemented in our tool that iterates over the TDL model to collect all *Gate Type* elements and generates for each a message-based port statement in TTCN-3 syntax. The *instantiate* elements are parsed to generate a component-type statement with an associated port.

III. Generate the Description Module

The TDL *Test Description* defines the ATC behavior, mainly composed of the actions and interactions exchanged between components over the communication gates. An action is used to specify a procedure (e.g. local computation, function call, physical setup, etc.) informally, whereas interactions refer to the data being exchanged between the components. In TTCN-3 realization, our tool iterates over the TDL model elements to parse the behavior elements and generate equivalent statements for each in TTCN-3. Our tool parses the *sends instance* statements (interaction) and generates a TTCN-3 message statement (Rule #8) as shown in **Figure 5.11**.

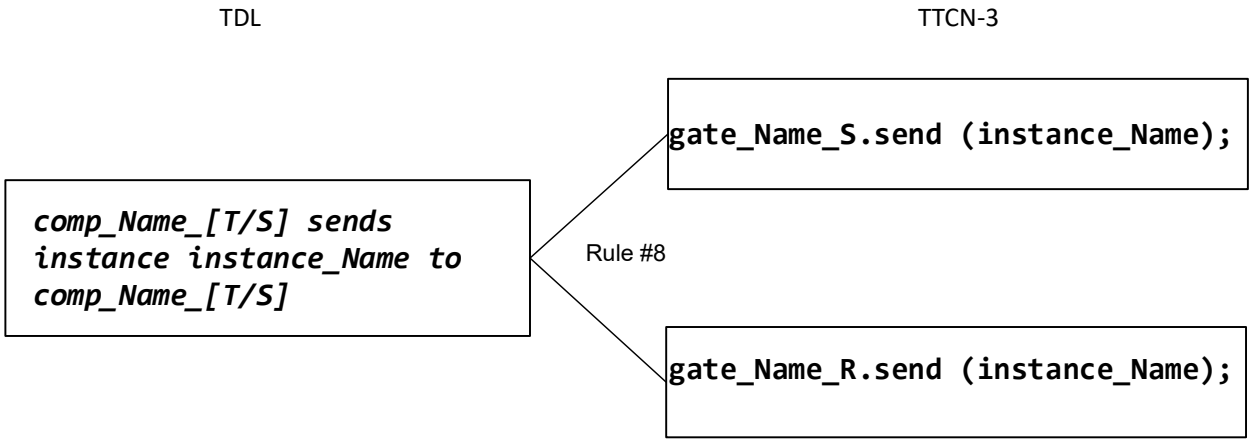


Figure 5.11: TDL interaction transformation

The *action* statement is parsed to generate a function signature and a function call (Rule #12) as shown in **Figure** 5.12.

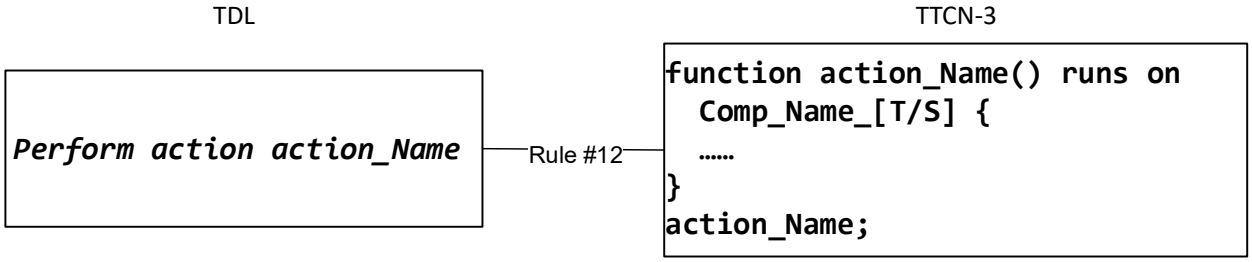


Figure 5.12: TDL Action transformation

The obtained function is refined at the TTCN-3 level when applicable. Other TDL behavioral statements are mapped to TTCN-3 constructs to be used in TCs or in functions by applying the corresponding rule.

5.3.1.4 The Completeness and Soundness of the Model Transformations

In general, model transformations are used between different domains for model evolution, code generation and analysis. The UCM models are adequate for describing the functional requirements of a system. Their automated transformation to TDL models bridged the gap with the TCs in TTCN-3. However, the metamodel of the exported scenario generated from UCM scenario, early in the process, doesn't have an alternative element that normally a TC has to handle alternate test behavior. The absence of an alternative element in the scenario

metamodel required post-processing of the generated TDL interaction behavior to merge those that constitute alternate test behaviors. The transformation of TDL models allowed refining and generating TCs that can be performed on the SUT. The model transformations here link the various test artifacts and promise to reduce the required amount of manual work for test development.

5.3.1.5 Test Case Generation Approach Feasibility

The feasibility of the approach is demonstrated via a case study from the avionics public domain called landing gear system (LGS) [136].

The LGS specifications are categorized into functional, safety and timing requirements. In the next sections, the behavior of the LGS is described from a Pilot's perspective, formalized into a given use case notation and then mapped to UCM scenario models. The LGS supports an aircraft when it is on the ground, allowing it to take off, land and taxi. Most modern aircraft have a retractable undercarriage, which folds away during the flight to reduce air resistance or drag. A conventional hydraulic LGS has a tricycle configuration consisting of the nose and the main (left and right) landing gears. Each landing configuration contains a door, the landing gear, and the associated hydraulic cylinders. The LGS is representative of critical ESs. Failure to deploy it puts the life of passengers in danger and causes massive airframe damage upon landing. Prior to landing, the landing sequence of an aircraft is: open the landing gearbox doors, extend the landing gear and close the doors. After taking off, the Retraction Sequence is: open the landing gearbox doors, retract the landing gear and close the doors. The LGS is composed of: (a) mechanical part; (b) digital part; and (c) a Pilot interface part which is further detailed in the next paragraph in order to identify the requirements. For more information about parts (a) and (b), please refer to [136].

The Pilot commands the retraction and extension of the gears by switching a handle up or down. When the handle is switched to "Up" the retracting landing gear sequence is executed, and when the handle is switched to "Down," the landing gear extension sequence is executed. Additionally, the Pilot's control panel has a set of lights indicating the current positions of the gears and doors, as well as the current health state of the system and its equipment. These lights and their indications are:

- One green light: “gears are locked down”.
- One amber light: “gears are in transition”.
- One red light: “landing gear system failure”.
- No light is ON: “gears are locked up”.
- Doors locked opened sign is ON: “all doors of the landing gearboxes are locked in opened position”.
- Doors locked opened sign is OFF: “all doors are unlocked”.
- Doors locked closed sign is ON: “all doors of the landing gearboxes are locked in closed position”.
- Doors locked closed sign is OFF: “all doors are unlocked”.
- Normal Mode Fail sign is ON: “Normal Mode Fail”.
- Normal Mode Fail sign is OFF “Normal Mode Pass”.

The expected behavior of the LGS is implemented by the control software whose aim is twofold: (1) control the hydraulic devices according to the Pilot’s orders and to the mechanical devices’ positions and (2) monitor the system and inform the Pilot in case of any malfunction.

Before showing how the functional and timing requirements of the LGS can be captured by UCM scenario models, the LGS requirements are formalized as described next.

I. Modeling LGS Requirements into Cockburn Use Case Notation

The LGS requirements fall into two basic scenarios: the Extending Sequence and the Retraction Sequence. For clarification, the Extending Sequence scenario, as defined in the case study, is used as a running example. Next, consider that the Pilot wants to land the airplane and so switches the handle down when the aircraft has an indicated airspeed of less than 200 knots and an altitude less than 2500 feet. The Extending Sequence scenario is written as a use case follows:

USE CASE: Extending Sequence.

Primary Actor: Pilot

Secondary Actor: Landing Gear Control Unit (LGCU)

Scope: LGS.

Precondition: Airspeed is less than 200 knots and altitude is less than 2500 feet.

Minimal guarantee: Landing gears are extended in emergency mode.

Success guarantee: Landing gears are extended in normal mode.

Trigger: Pilot switches handle down.

Main success scenario:

1. Pilot switches handle down and it stays down.
2. LGCU activates doors opening.
3. LGCU locks door in opened position.
4. LGCU switches doors locked open sign to ON
5. LGCU releases up-lock gears.
6. LGCU switches amber light to ON.
7. LGCU locks down gears when they reach the full-down position.
8. LGCU switches green light to ON and amber light to OFF.
9. LGCU activates doors closing.
10. LGCU locks door in closed position.
11. LGCU switches doors locked closed sign to ON

12. Pilot confirms the successful deployment of the landing gears.

Extensions: (Failure mode)

1.a If the landing gear command handle has been DOWN for 15 seconds and the gears are not locked down after 15 s, then the LGCU switches red light to ON (failure in deployment).

2.aIf one of the three doors are still seen locked in the closed position more than 7 seconds after activating doors opening, then the LGCU fails Normal Mode.

3.aIf one of the three doors are not seen locked in the opened position more than 7 seconds after activating doors locking in opened position, then the LGCU fails Normal Mode.

5.a If one of the three gears are still seen locked in the up position more than 7 seconds after releasing the up-lock, then the LGCU fails Normal Mode.

9.a If one of the three gears are not seen locked in the down position more than 10 seconds after releasing the up-lock, then LGCU fails Normal Mode. If one of the three doors are still seen locked in the opened position more than 7 second after activating doors closing, then the LGCU fails Normal Mode.

10.a If one of the three doors are not seen locked in the closed position more than 7 seconds after activating doors locking, then LGCU fails Normal Mode.

Next, we proceed with the mapping of the Extending Sequence use case to UCM scenario models.

II. Mapping LGS Use Case to UCM Scenario Models

UCM scenario models can be built by mapping the actors and the actions elements defined in the Extending Sequence use case. The mapping is straightforward, for example, the Primary Actor (Pilot) and the Secondary Actor (LGCU) are mapped manually to two UCM components: *Pilot* and *LGCU*. The actions to be performed by each component, such as *Handle*_Down and *ReleaseUp_Lock* are allocated to UCM *responsibility* elements. As a rule, each action in the use

case is mapped to one *responsibility* element in UCM. As a result, two lists of *responsibility* are extracted from the use case and bound to their corresponding components:

- *Pilot: { Handle_Down* and *ConfirmGearsDown}*
- *LGCU: {OpenDoors, LockDoorsInOpenedPos, ReleaseUp_Lock, AmberON, Lock_DownGears, GreenON_AmberOFF, CloseDoors, LockDoorsInClosedPos, RedON,* and *NormalModeFailed}*.

With some basic knowledge of the jUCMNav tool, the two lists of responsibility; *Pilot* and *LGCU*, along with timed requirements in the use case can be modeled into UCM scenarios. **Figure** 5.13 shows a UCM map that is composed of two components with their bounded responsibilities. The time constraints and functional requirements are modeled as indicated by the Extending Sequence use case. The map in **Figure** 5.13 encloses eight possible scenario models representing the Extending Sequence requirements of the LGS.

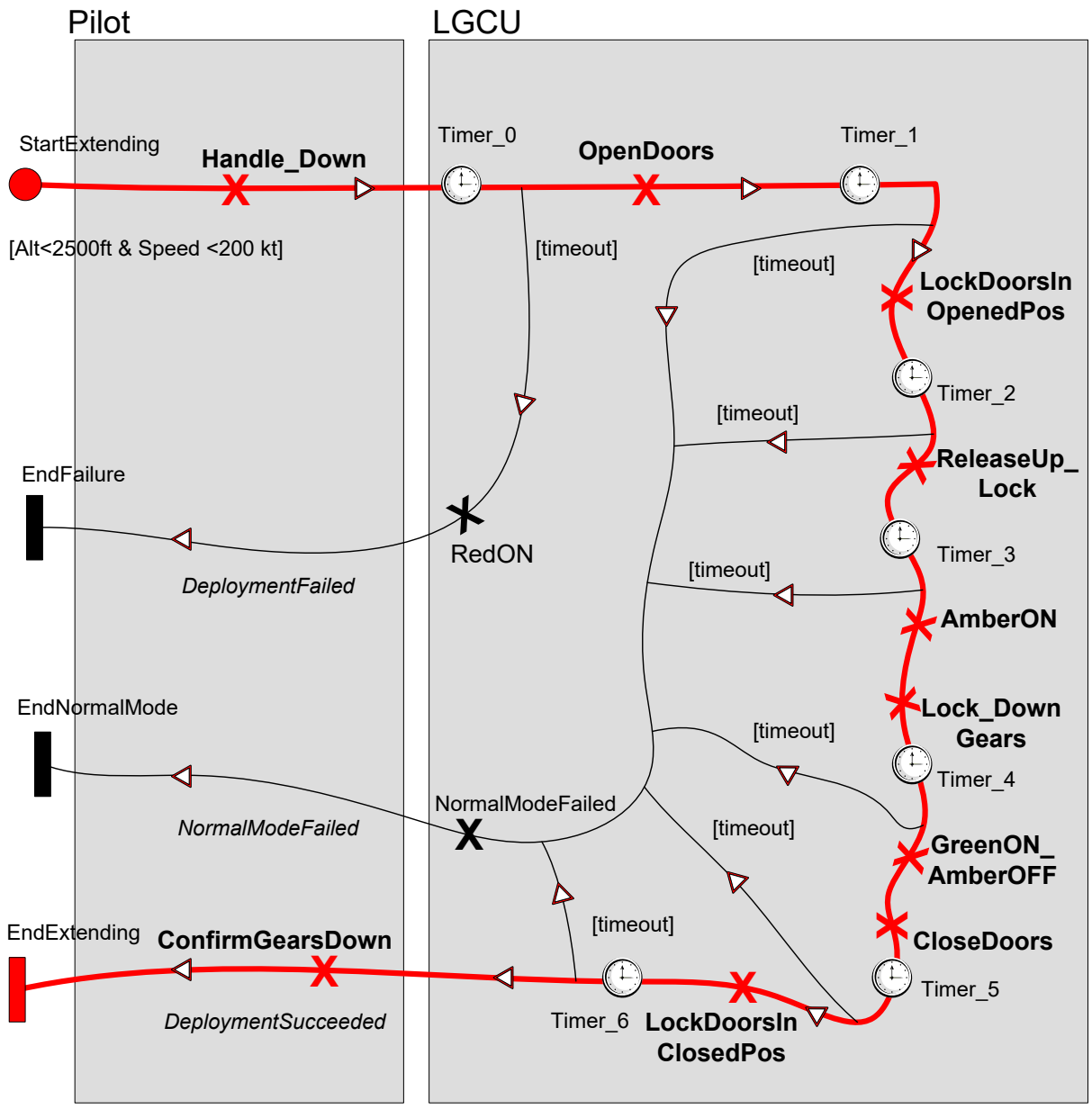


Figure 5.13: UCM scenario models built from an Extending Sequence use case

These scenario models fall into three major groups:

a) Successful Deployment Group: contains one scenario model, labeled "*DeploymentSucceeded*".

b) Gears Deployment Failed Group: contains one scenario model, labeled "*DeploymentFailed*".

c) Normal Mode Failed Group: contains six scenario models, all of them end in the path labeled "*NormalModeFailed*".

The execution of any of the scenario models begins at the *StartExtending* point (filled circle) and terminates in one of the three End points (bars); *EndExtending*, *EndNormalMode* or *EndFailure*. The *StartExtending* point is triggered when its preconditions are met— the airplane achieves airspeed of less than 200 *knots* and altitude below 2500 *feet*. The *Pilot* then switches the *Handle_Down* causing the *LGCU* to extend the landing gears scenario.

In this exercise of creating UCM scenario models, the Extending Sequence requirements of the LGS are developed and allocated to software items.

In the next section, we show how the "*DeploymentSucceeded*" scenario model is transformed into an ATC.

III. Transform UCM Scenario Models and Data Model into ATC in TDL

We explain in detail in the following subsections how each element in the TDL specification is developed in the *ATC Builder* process.

— Generate TDL Test Objective

In our experimentation, the TDL *Test Objectives* shown in Listing 5-1 were developed manually by analyzing the sequence and role of UCM objects that reside on the "*DeploymentSucceeded*" scenario and enriched with test requirements.

```
1.  Test Objective TestObj1 {
2.  description: "ensure that when Handle is switched down, a timer is started. If it times-out 15 seconds later
3.  and  gears are not locked, a red light is sent"; }
4.  Test Objective TestObj2 {
5.  description: "ensure that a 'door locked open light' is received after locking the doors in opened position"; }
6.  Test Objective TestObj3 {
7.  description: "ensure that an 'amber light' is received when gears are in transition." ; }
8.  Test Objective TestObj4 {
9.  description: "ensure that a 'green light' is received when gears are locked down." ; }
10. Test Objective TestObj5 {
11. description: "ensure that a 'door locked close light' is received after closing the door";  }
```

Listing 5-1: TDL Test Objective

— Generate TDL Data Set

In the "*DeploymentSucceeded*" scenario, the *Pilot* and *LGCU* components interact with each other through stimuli and responses. For example, the *Pilot* sends a stimulus to the *LGCU* when executing *Handle_Down responsibility*. The *LGCU* responds by performing internal actions (no interaction) when executing *OpenDoors* and *ClosedDoors responsibilities* and sending responses when stepping into *LockDoorsInOpenedPos*, *AmberON*, *GreenON_AmberOFF*, and *LockDoorsInClosedPos responsibilities*. **Table** 5-3 shows the test data for the UCM "*DeploymentSucceeded*" scenario.

Table 5-3: Test Data For "*DeploymentSucceeded*" Scenarion

Test Data Requirement	UCM responsibility Stimulus/Response	TDL Data Instances
Stimulus to be sent when Pilot switches handle down	*Handle_Down*	instance *Handle_Down*
Response to be received when LGCU locks doors in opened position	*LockDoorsInOpenedPos*	instance *LockDoorsInOpenedPos*
Response to be received when LGCU activates Gear maneuvering	*AmberON*	instance *AmberON*
Response to be received when LGCU locks Gears in a down position	*GreenON_AmberOFF*	instance *GreenON_AmberOFF*
Response to be received when LGCU locks doors in closed position	*LockDoorsInClosedPos*	instance *LockDoorsInClosedPos*

The developed TDL *Data Instances* are grouped in two *Data Set* elements in terms of Stimulus and Response:

- **GearDeployment:** bounded to *Pilot* messages (Stimulus); and
- **Signal:** bounded to *LGCU* messages (Response).

Listing 5-2 shows compiled TDL *Data Instances* grouped in two *Data Sets* that are developed from test data in **Table** 5-3.

```
1.  Data Set GearDeployment {
2.    instance Handle_Down;
3.  }
4.  Data Set Signal  {
5.    instance LockDoorsInOpenedPos;
6.    instance AmberON;
7.    instance GreenON_AmberOFF;
8.    instance LockDoorsInClosedPos;
9.   }
```

Listing 5-2: TDL Data Sets elements

— Generate TDL Test Configuration

The UCM "*DeploymentSucceeded*" scenario in **Figure** 5.13 is exported, using the UCM *traversal mechanism* [133], to a scenario that contains traversed UCM elements. A snapshot of the exported scenario that highlights the *Test Configuration* is shown in Listing 5-3. In this exportation, the UCM components Pilot and LGCU are mapped to TDL *Component Instance* objects with a *Gate Instance*. A *Connection* instance is added to indicate that the two *Component Instances* should be connected.

```
1.  <?xml version="1.0" encoding="ISO-8859-1"?>
2.  <tdl:Package xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns:tdl="http://www.etsi.org/spec/TDL/20130606" name="SuccessfulDeployment">
3.  <comment name="Created" body="April 16, 2016 11:10:12 AM EDT"/>
4.  <comment name="Modified" body="April 16, 2016 11:10:12 AM EDT"/>
5.  <comment name="Author" body="nkesserw"/>
6.  <packagedElements xsi:type="tdl:TestConfiguration">
7.  <componentInstance name="Pilot" type="//@packagedElements.18">
8.  <gateInstance name="gPilot" type="//@packagedElements.6"/>
9.  </componentInstance>
10. <componentInstance name="LGCU" type="//@packagedElements.19">
11. <gateInstance name="gLGCU" type="//@packagedElements.6"/>
12. </componentInstance>
13. <connection name="LGCU_Pilot" endPoint="//@packagedElements.0/@componentInstance.2/@gateInstance.0
    //@packagedElements.0/@componentInstance.1/@gateInstance.0"/>
14. </packagedElements>
```

Listing 5-3: A snapshot of the exported "*DeploymentSucceeded*" scenario that shows the TDL Test Configuration package

Next, a compiled *Test Configuration* element is achieved by parsing the exported scenario to convert the packaged element *tdl:TestConfiguration* into concrete TDL syntax. The *Component Instances* are instantiated to either SUT or Tester, depending on their role. Each *Component Instance* has a gate type to specify the data that can be exchanged, i.e., the *Data Sets* developed earlier. Listing 5-4 shows the TDL *Test Configuration* generated automatically from the exported "*DeploymentSucceeded*" scenario depicted in Listing 5-3. The *Data Sets GearDeployment* and *Signal* are added manually to the TDL *Test Configuration* (line 1). The two components: *Pilot* and *LGCU* are typed (line 7 and line 10) and connected through newly-defined gates (line 13).

```
1.  Gate Type defaultGT accepts GearDeployment, Signal;
2.  Component Type defaultCompType {
3.    gate types :defaultGT ;
4.  }
5.  Test Configuration TestConfiguration {
6.     //Pilot component
7.    instantiate Pilot as Tester of type defaultCompType having {
8.    gate gPilot of type defaultGT ; }
9.   //LGCU component
10.   instantiate LGCU as SUT of type defaultCompType having {
11.    gate gLGCU of type defaultGT ;  }
12.  / /connect the two components through their gates
13.  connect gPilot to gLGCU;  }
```

Listing 5-4: TDL Test Configuration element generated from a "DeploymentSucceeded" scenario

— Generate TDL Test Description

The TDL *Data Instances* shown in Listing 5-2 are used as *Interaction* objects between a Tester and an SUT. The UCM *responsibility* objects along the "*DeploymentSucceeded*" scenario is mapped to the *Action Reference*. UCM scenario Timer Set events are mapped to *TimerStart* objects in TDL. Listing 5-5 shows a snapshot of the exported "*DeploymentSucceeded*" scenario.

```
1.  <packagedElements xsi:type="tdl:TestDescription" name="TestSuccessfulDeployment"
    testConfiguration="//@packagedElements.0">
2.  <behaviour>
3.  <block>
4.  <behaviour xsi:type="tdl:ActionReference" name="Handle_Down" action="//@packagedElements.22">
5.  <annotation value="gPilot" key="//@packagedElements.5"/>
6.  </behaviour>
7.  <behaviour xsi:type="tdl:Interaction" name="From Pilot to LGCU"
    source="//@packagedElements.0/@componentInstance.1/@gateInstance.0"
    target="//@packagedElements.0/@componentInstance.2/@gateInstance.0">
8.  <annotation value="Timer_0" key="//@packagedElements.1"/>
9.  <annotation value="If we had a description for this Interaction we could put it here."
    key="//@packagedElements.2"/>
10. </behaviour>
11. <behaviour xsi:type="tdl:TimerStart" name="Timer_0 _Start" timer="//@packagedElements.19/@timer.0">
12. <annotation value="gLGCU" key="//@packagedElements.3"/>
13. </behaviour>
14. <behaviour xsi:type="tdl:TimerStop" name="Timer_0 _TimerStop"
    timer="//@packagedElements.19/@timer.0">
15. <annotation value="gLGCU" key="//@packagedElements.3"/>
16. </behaviour>
17. <behaviour xsi:type="tdl:AlternativeBehaviour" name="OrFork1291\nisGearsDown">
18. <annotation value="gLGCU" key="//@packagedElements.4"/>
19. </behaviour>
20. <behaviour xsi:type="tdl:ActionReference" name="OpenDoors" action="//@packagedElements.23">
21. <annotation value="gLGCU" key="//@packagedElements.5"/>
22. </behaviour>
23. </block>
24. </behaviour>
```

Listing 5-5: A snapshot of the exported "*DeploymentSucceeded*" scenario that shows the TDL Test Description package

Developing a TDL *Test Description* is automated by parsing the exported "*DeploymentSucceeded*" scenario, extracting the components with their bounded *responsibilities* and mapping them to equivalent TDL objects. Listing 5-6 shows the TDL *Test Description* that is composed of actions, timers and interactions. As mentioned earlier, the absence of alternative elements in the scenario metamodel required post-processing of the generated *Test Description* to

merge the scenarios that constitute alternate test behaviour. The element *repeat* iterates over the different alternatives a number of times as determined by the 'numIteration' attribute.

```
1.  Test Description TestDescription {  //Test description definition
2.   use configuration : TestConfiguration;  {
3.   perform action Handle_Down on component Pilot with {  PRECONDITION ;  };
4.   gPilot sends instance Handle_Down to gLGCU with { test objectives :TestObj1;   };
5.   perform action OpenDoors on component LGCU with {  PRECONDITION ;  };
6.   perform action LockDoorsInOpenedPos on component LGCU with {PRECONDITION ;} ;
7.   repeat 4 times {  //Iterate over receiving responses, each one is consumed once
8.     alternatively  { // LGCU sends response indicating Door is locked in open position
9.       gLGCU sends instance LockDoorsInOpenedPos to gPilot with
10.      { test objectives : TestObj2; };
11.      set verdict to PASS ; }
12.    or { gate gLGCU  is quiet for (7.0 SECOND);
13.      set verdict to FAIL;  }
14.      perform action ReleaseUp_Lock on component LGCU with {  PRECONDITION; };
15.   alternatively  {  // LGCU sends response indicating Gears are in transition
16.     gLGCU sends instance AmberON to gPilot with { test objectives : TestObj3; };
17.     set verdict to PASS ;    }
18.   or { gate gLGCU is quiet for (7.0 SECOND);
19.     set verdict to FAIL;  }
20.     perform action Lock_DownGears on component LGCU with { PRECONDITION ; };
21.   alternatively  { // LGCU sends response indicating Gears are in locked down
22.     gLGCU sends instance GreenON_AmberOFF to gPilot with { test objectives : TestObj4;  };
23.     set verdict to PASS ; }
24.   or { gate gLGCU is quiet for (7.0 SECOND);
25.     set verdict to FAIL;    }
26.     perform action CloseDoors on component LGCU;
27.     perform action LockDoorsInClosedPos on component LGCU with  {
28.                     PRECONDITION; };
29.  alternatively  { // LGCU sends response indicating Door is locked in close position
30.    gLGCU sends instance LockDoorsInClosedPos to gPilot with {test objectives :TestObj5; };
31.    set verdict to PASS ;
32.    perform action ConfirmGearsDown on component Pilot with {PRECONDITION ;};}
33.  or { gate gLGCU is quiet for (7.0 SECOND) ;
34.     set verdict to FAIL;  }
35.  or { gate gLGCU is quiet for (15.0 SECOND);
36.     set verdict to FAIL;  }
37.   }
38.  } }
```

Listing 5-6: TDL Test Description element generated from "*DeploymentSucceeded*" scenario

The interactions of the *Test Description* start when a *Handle_Down* command flows from the *Pilot* gate to the *LGCU* gate (line 4). Immediately afterward, a timer is started to satisfy the timing constraint of the landing gears' deployment, followed by a second timer to time the action of the door opening. Shortly after locking the doors in the opened position, the *LGCU* gate sends the *LockDoorsInOpenedPos* sign (line 9) indicating all the doors are locked in the opened position. The *LGCU* releases the *up-lock* and an *AmberON* status is sent (line 16) indicating the gears are in transition to the full-down position. Another timer is started to time the action of locking the gears in the down position. Gears are locked once they reach the final position when a *GreenON_AmberOFF* status message is sent from the *LGCU* gate (line 22) indicating full deployment of the landing gears. If the *GreenON_AmberOFF* status message sign is received before any time expiration, a pass verdict is issued and the *Pilot* confirms gears are down and locked (line 32), otherwise the test fails.

The elements obtained earlier— *Test Objective*, *Data Set*, *precondition* and *Test Configuration*— are used in the *Test Description* to help structure the TDL Specification. Listing 5-7 shows the developed TDL Test Specification. In the next section, we show how to script the obtained TDL specification into TCs in TTCN-3.

```
TDLan Specification DeployLandingGearTest {
 Verdict PASS; Verdict FAIL;
 Action Handle_Down: "when airspeed is less than 200 knots and altitude is less than 2500 feet, the pilot switches handle down and keep it down for 15
                      seconds, gears starts";
 Action OpenDoors: "when doors are locked in closed position, the corresponding cylinder are extended to unlock the doors";
 Action LockDoorsInOpenedPos: "lock the doors in opened position";
 Action ReleaseUp_Lock: "when gears are locked in up position, the gear cylinders receive hydraulic pressure in order to release the lock that holds the
                         gears";
 Action Lock_DownGears: " lock gears when reach full down position";
 Action CloseDoors: "when doors are locked in opened position, the corresponding cylinder are extended to unlock the doors";
 Action LockDoorsInClosedPos: "lock the doors in closed position";
 Action ConfirmGearsDown: "Pilot confirms gears are down and locked";
 Annotation PRECONDITION ;
 Time Unit SECOND;
 Test Objective TestObj1 {
 description: "ensure that when Handle is switched Down, a timer is started. If it times-out 15 seconds later and gears are not locked, a red light is sent";}
 Test Objective TestObj2 {
 description: "ensure that a 'door locked open light' is received after locking the doors in opened position.";}
 Test Objective TestObj3 {
 description: "ensure that an 'amber light' is received when gears are in transition.";}
 Test Objective TestObj4 {
 description: "ensure that a 'green light' is received when gears are locked down.";}
 Test Objective TestObj5 {
 description: "ensure that a 'door locked close light' is received after closing the door.";}
 Data Set GearDeployment {
 instance Handle_Down; }
 Data Set Signal { instance LockDoorsInOpenedPos;  instance AmberON;  instance GreenON_AmberOFF;  instance LockDoorsInClosedPos; }
 //Data Instance reference
 Use "LandingGearData.ttcn3" as LGearData;
 Map Handle_Down to " Handle_DownTemplate" in LGearData;
 Map LockDoorsInOpenedPos to " LockDoorsInOpenedPosTemplate" in LGearData;
 Map AmberON to " AmberONTemplate" in LGearData;
 Map GreenON_AmberOFF to " GreenON_AmberOFFTemplate" in LGearData;
 Map LockDoorsInClosedPos to " LockDoorsInClosedPosTemplate" in LGearData;
 Gate Type defaultGT accepts GearDeployment, Signal; //Define the gate type and the exchanged data set
 Component Type defaultCompType {  gate types :defaultGT ; }
 Test Configuration TestConfiguration  { // Pilot and LGCU
   instantiate Pilot as Tester of type defaultCompType having { gate gPilot of type defaultGT ;  }
   instantiate LGCU as SUT of type  defaultCompType having{ gate gLGCU of type  defaultGT ;   }
  connect gPilot to gLGCU;  }   //connect the two components through their gates
 Test Description TestDescription {  //Test description definition
   use configuration : TestConfiguration;  {
   perform action Handle_Down on component Pilot with {  PRECONDITION ;  };
   gPilot sends instance Handle_Down to gLGCU with { test objectives :TestObj1;   };
   perform action OpenDoors on component LGCU with {  PRECONDITION ;  };
   perform action LockDoorsInOpenedPos on component LGCU with {  PRECONDITION ;  };
   repeat 4 times {   //Iterate over receiving responses, each one is consumed once
     alternatively  { // LGCU sends response indicating Door is locked in opened position
        gLGCU sends instance LockDoorsInOpenedPos to gPilot with { test objectives : TestObj2; }; set verdict to PASS ; }
     or { gate gLGCU  is quiet for (7.0 SECOND); set verdict to FAIL;  }
        perform action ReleaseUp_Lock on component LGCU with {  PRECONDITION ;  };
     alternatively  { // LGCU sends response indicating Gears are in transition
        gLGCU sends instance AmberON to gPilot with { test objectives : TestObj3; }; set verdict to PASS ;    }
     or { gate gLGCU is quiet for (7.0 SECOND); set verdict to FAIL;  }
        perform action Lock_DownGears on component LGCU with {  PRECONDITION ;  };
     alternatively  { // LGCU sends response indicating Gears are locked down
        gLGCU sends instance GreenON_AmberOFF to gPilot with { test objectives : TestObj4;  };  set verdict to PASS ; }
     or { gate gLGCU is quiet for (7.0 SECOND); set verdict to FAIL;    }
        perform action CloseDoors on component LGCU;
        perform action LockDoorsInClosedPos on component LGCU with {  PRECONDITION ;  };
     alternatively  {  // LGCU sends response indicating Door is locked in closed position
       gLGCU sends instance LockDoorsInClosedPos to gPilot with { test objectives : TestObj5; }; set verdict to PASS ;
       perform action ConfirmGearsDown on component Pilot with {  PRECONDITION ;  }; }
     or { gate gLGCU is quiet for (7.0 SECOND) ; set verdict to FAIL;  }
     or { gate gLGCU is quiet for (15.0 SECOND);
        set verdict to FAIL;  }
   }
  }
 }
}
```

Listing 5-7: The resulting TDL specification model

IV. Transform TDL Specifications to TTCN-3 Modules

In the following subsections, we show how the TTCN-3 modules are developed from the TDL Specification.

— Generate TTCN-3 Test Data

The two *Data Sets; GearDeployment* and *Signal*, defined previously in Listing 5-2, are parsed with their *instances* to generate records and record fields (variables) in TTCN-3 syntax based on Rule #22 and Rule #23. After the TTCN-3 data module is partially generated and test data becomes available, the module is completed with test oracle information and typed with concrete TTCN-3 types. **Figure** 5.14 shows the transformation (semi-automatic) between TDL *Data Sets* and TTCN-3 data module.

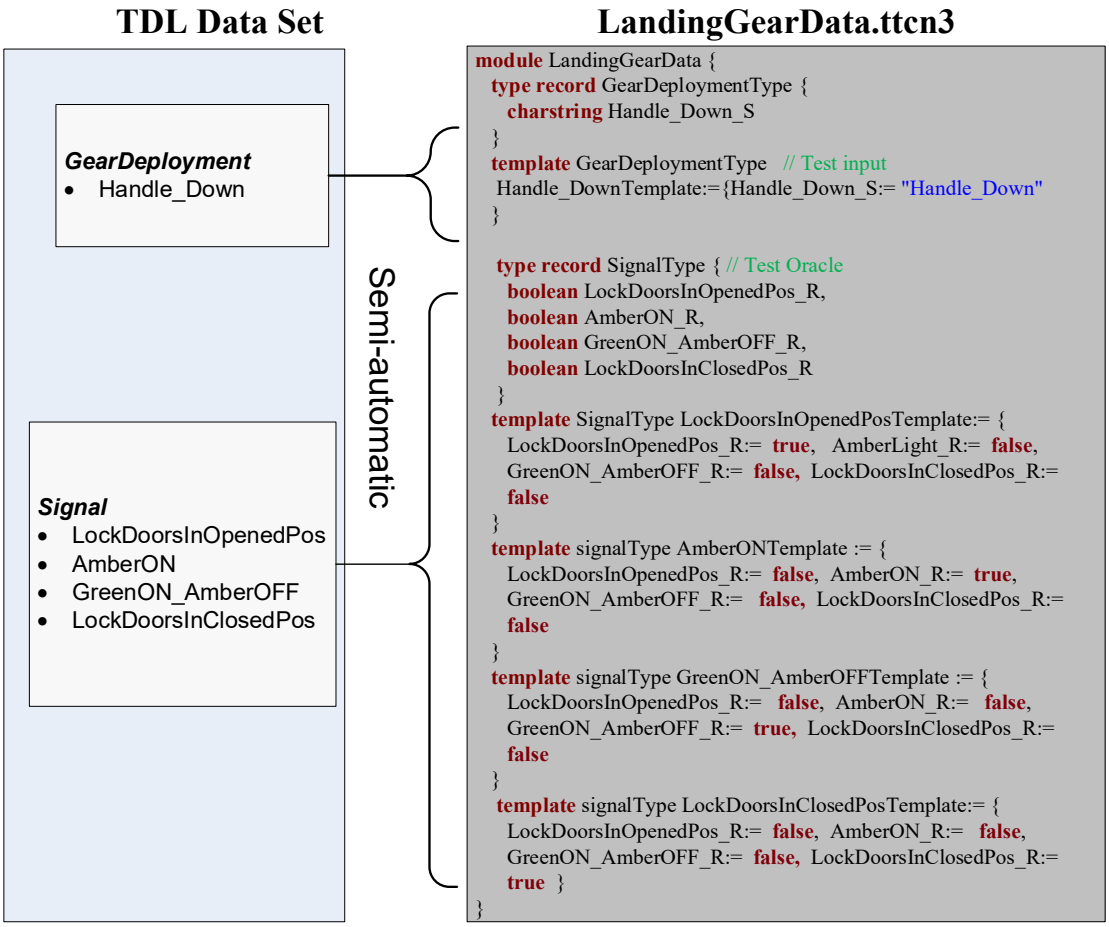


Figure 5.14: Mapping abstract TDL Data Sets to concrete data in TTCN-3

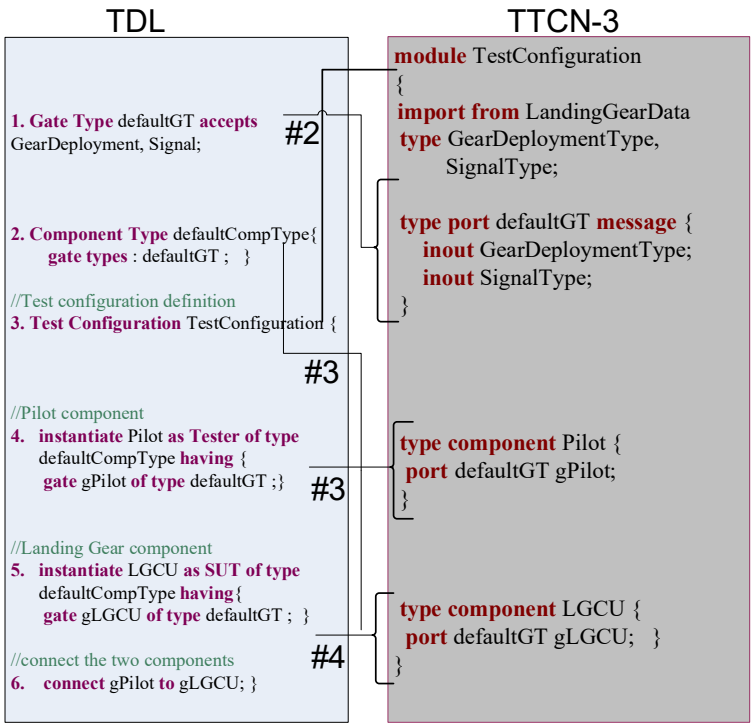
The *Data instances* developed in the previous section are next mapped to the corresponding TTCN-3 templates through TDL data element mappings as shown in Listing 5-8.

```
1. Use "LandingGearData.ttcn3" as LGearData;
2. Map Handle_Down to " Handle_DownTemplate" in LGearData;
3. Map LockDoorsInOpenedPos to " LockDoorsInOpenedPosTemplate" in LGearData;
4. Map AmberON to " AmberONTemplate" in LGearData;
5. Map GreenON_AmberOFF to " GreenON_AmberOFFTemplate" in LGearData;
6. Map LockDoorsInClosedPos to " LockDoorsInClosedPosTemplate" in LGearData;
```

Listing 5-8: TDL Map elements used to reference concrete TTCN-3 templates

— Generate TTCN-3 Test Configuration

Based on the transformation rules; Rule #2, Rule #3, and Rule #4, the transformation of the obtained TDL *Test Configuration* into an equivalent one in TTCN-3 is performed. Listing 5-9 shows the transformation of one TDL *Test Configuration* into an equivalent one in TTCN-3.



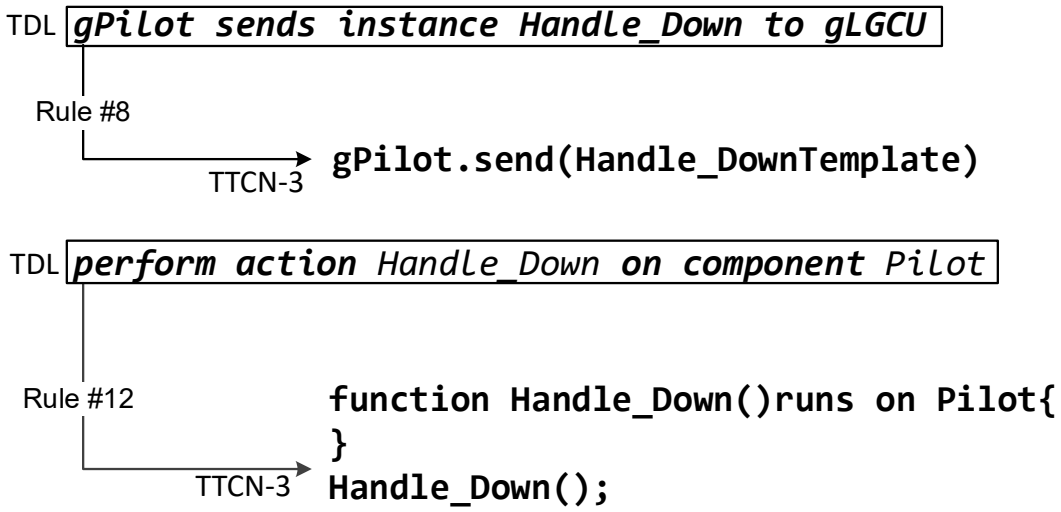
Listing 5-9: Transformation of TDL Test Configuration to its corresponding TTCN-3

The obtained Test Configuration in TTCN-3 defines test component types and port types, denoted by the keyword's **component** and **port**. The communication between the components is achieved via the message-based communication port g*Pilot* and *gLGCU,* through which

messages of type *GearDeploymentType* and *signalType* are sent and received. The connection between the two components is shown in the Test Description module (developed next) and is expressed with a map function.

— Generate TTCN-3 Test Behavior

As mentioned previously, the developed tool maps the TDL elements to TTCN-3 statements. Listing 5-10 shows an example of transforming two major TDL elements: *action* and *interaction*. The tool parses the *sends instance* statements (interaction) and generates a TTCN-3 message statement. The *action* statement is parsed to generate a function signature and a function call. The obtained function is refined at the TTCN-3 level when applicable.



Listing 5-10: TDL action and interaction transformation

The transformation of TDL Test Objectives cannot be rule-based. However, their semantics can be interpreted manually and reflected in the TTCN-3 Test Description module. Listing 5-11 shows TTCN-3 Test Description module transformed from the TDL Specification.

```
1.  module TestDescription {
2.    import from TestConfiguration all;
3.    import from LandingGearData all;
4.    testcase _TC () runs on Pilot {
5.    map (mtc:gPilot, system:gLGCU);
6.    timer deploymentTime; timer lockDoorOpenedTime;   timer gearsManeouvringTime;
7.    timer gearLockedDownTime;  timer lockDoorClosedTime;
8.    Handle_Down();  // function call
9.    gPilot.send(Handle_DownTemplate);
10.   deploymentTime.start(15.0);
11.   OpenDoors();  // function call
12.   LockDoorsInOpenedPos ();
13.   lockDoorOpenedTime.start(7.0);
14.   alt {
15.   [] gPilot.receive(LockDoorsInOpenedPosTemplate) {
16.      lockDoorOpenedTime.stop;
17.      setverdict(pass);
18.     ReleaseUp_Lock();  // function call
19.     gearsManeouvringTime.start(7.0);
20.     repeat  } // restart the alt
21.   [] lockDoorOpenedTime.timeout {
22.     setverdict(fail) }
23.   [] gPilot.receive(AmberONTemplate) {
24.     gearsManeouvringTime.stop;
25.     setverdict(pass);
26.     Lock_DownGears();   // function call
27.     gearLockedDownTime.start(7.0);
28.     repeat    } // restart the alt
29.   [] gearsManeouvringTime.timeout {
30.     setverdict(fail) }
31.   [] gPilot.receive(GreenON_AmberOFFTemplate) {
32.     gearLockedDownTime.stop;
33.     setverdict(pass);
34.     CloseDoors();  // function call
35.     LockDoorsInClosedPos();
36.    lockDoorClosedTime.start(7.0);
37.    repeat } // restart the alt
38.   [] gearLockedDownTime.timeout {
39.     setverdict(fail) }
40.   [] gPilot.receive(LockDoorsInClosedPosTemplate) {
41.     lockDoorClosedTime.stop;
42.    deploymentTime.stop;
43.    setverdict(pass);
44.    ConfirmGearsDown(); } // function call
45.   [] lockDoorClosedTime.timeout {
46.     setverdict(fail) }
47.   [] deploymentTime.timeout {
48.     setverdict(fail) }  }
49.   unmap (mtc:gPilot, system:gLGCU);  }  }
50.   function Handle_Down () runs on Pilot {  }
51.   function OpenDoors () runs on Pilot {  }
52.   function LockDoorsInOpenedPos () runs on Pilot {   }
53.   function ReleaseUp_Lock () runs on Pilot {  }
54.   function Lock_DownGears () runs on Pilot {  }
55.   function CloseDoors () runs on Pilot {    }
56.   function LockDoorsInClosedPos () runs on Pilot {     }
57.   function ConfirmGearsDown () runs on Pilot {    }  }
```

Listing 5-11: TTCN-3 Test Description module

Now, the ETCs is completed by combining the derived modules represented by the three TTCN-3 files: "LandingGearData.ttcn3", "TestConfiguration.ttcn3", and "TestDescription.ttcn3". Listing 5-12 shows an additional module "DeployLandingGears.ttcn3" to invoke the TC execution.

```
1. module DeployLandingGears {
2.  import from TestDescription testcase _TC;
3.  control { execute(_TC());  }
4. }
```

Listing 5-12: TTCN-3 module to invoke the execution of the test case

5.3.2 Traceability Links Framework

The variety of different models produced in the TCG process discussed in the previous section poses challenges to requirements traceability and assessment. This diversity of artifacts results in an intricate relationship between requirements and the various models. The role played by relationships among artifacts to support automation of testing activities had long been recognized; relationships from behavioral models to test cases and from test cases to test results support coverage measurement, result evaluation and selective regression testing. The creation and maintenance of explicit relationships among test-related artifacts is, therefore the main challenge to the automated support of such activities.

In DO-178C, the software verification process defines activities for determining that the software aspects of airborne systems comply with airworthiness requirements. One of the activities defined in the process is to verify that the system requirements allocated to software have been developed into HLR that satisfy those system requirements. Trace data should be generated to support this verification. A relationship between each unique system-level requirement and its embodiment in the software requirement should be created, allowing traceability between software requirements and HLR. This relationship should allow for bidirectional traceability, meaning that the traceability chains can be traced in both the forwards and backward directions.

The rest of this section is structured as follows. Section 5.3.2.1 presents the traceability approach. A case study to demonstrate the approach realization is presented in Section 5.3.2.2.

5.3.2.1 Traceability Approach

In this section, we answer the RQ4: "*how to align the activities of requirement traceability to testing to improve project cost and comply with DO-178C standards?*" by presenting a framework that aligns the activities of requirement traceability to testing to improve system quality and project cost. The framework extends the MDTGL methodology to create explicit relationships in a trace model among testing artifacts. Our contribution is to build a traceability model to support the creation and persistence of relationships among these testing models. Moreover, to enable the support for visualizing traceability, model-based coverage analysis, and result evaluation. The approach relates UCM behavioral models to test cases via ATC models during model transformation where n-ary links among models could be visualized. This is an important factor in visualizing relationships among models because it is almost impossible to represent more than one link in a two-dimensional traceability matrix in an understandable way. Moreover, the number of relationships in traceability matrixes is high and fixed.

Figure 5.15 shows an overview of the approach. The first step in the approach from the traceability perspective is to create the UCM scenario model (step 1 in the figure). Then, the model is flattened to scenario definitions where each scenario is transformed to ATC in test description language TDL (step 2 in the figure). During this transformation, the traceability information is made explicit into a separate model. Then, (step 3 in the figure) TCG takes place; it consists of using the ATC model and data model to generate the test cases. Again, during the test cases generation, the traceability information, guided by a traceability scheme, is made explicit and persistent.

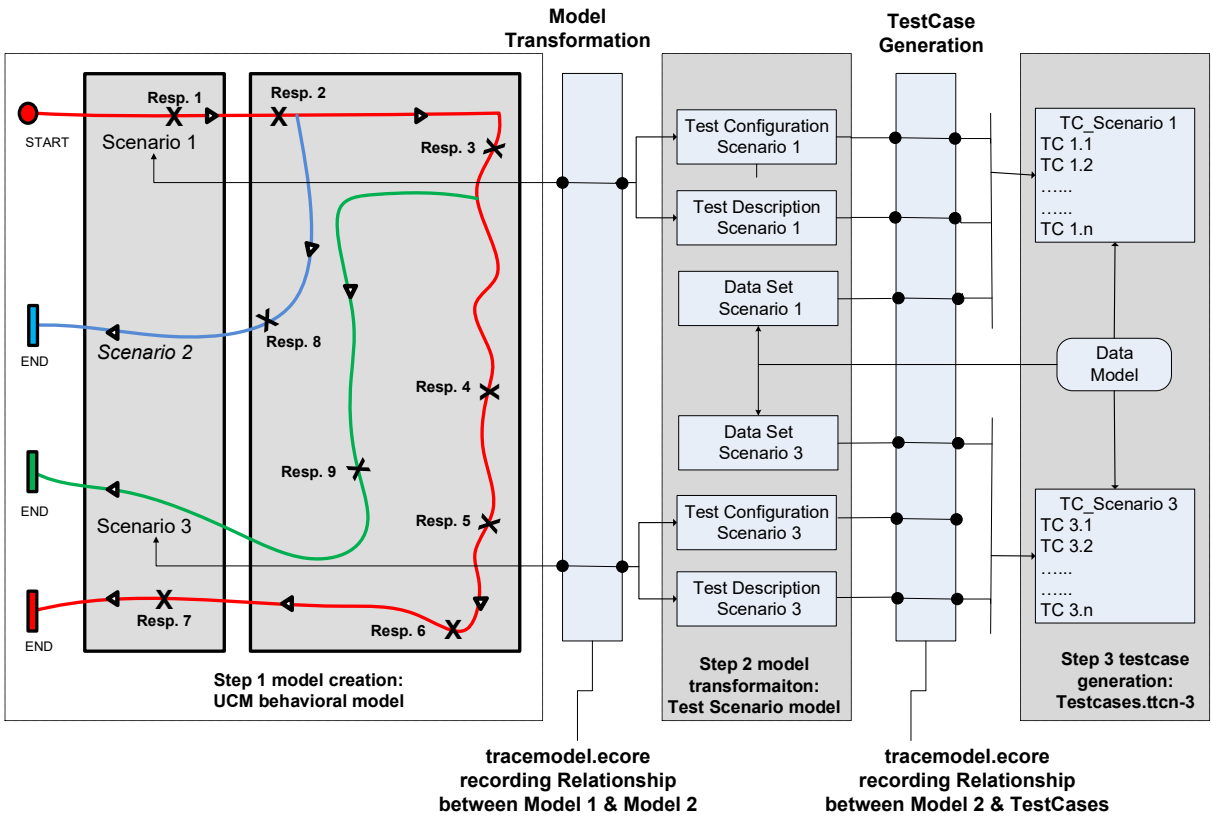


Figure 5.15: Traceability approach overview

During the execution of MDTGL methodology, the traceability information, recorded by our developed tools, is made explicit into a separate model called "*tracemodel.ecore*". The Ecore trace model records a small number of relationships from model to a test case to enable the support for model-based coverage analysis, visualizing traceability and result evaluation. Our Ecore[5] trace model is integrated into Eclipse Modeling Framework (EMF) and it is independent of the models it connects.

Our approach currently uses a trace metamodel inspired by Jouault et al. [137] that supports traceability. Our contribution is to externalize the relationships among the test-artifact models (UCM scenario models, ATCs models and ETCs models) and recorded them in our trace model. The relationships are created and recorded in the trace model to support activities such as result

[5] Ecore is the meta metamodel of *Eclipse Modeling Framework* (EMF). http://www.eclipse.org/modeling/emf/

evaluation, regression testing, and coverage analysis. The traceability metamodel is shown in **Figure** 5.16.

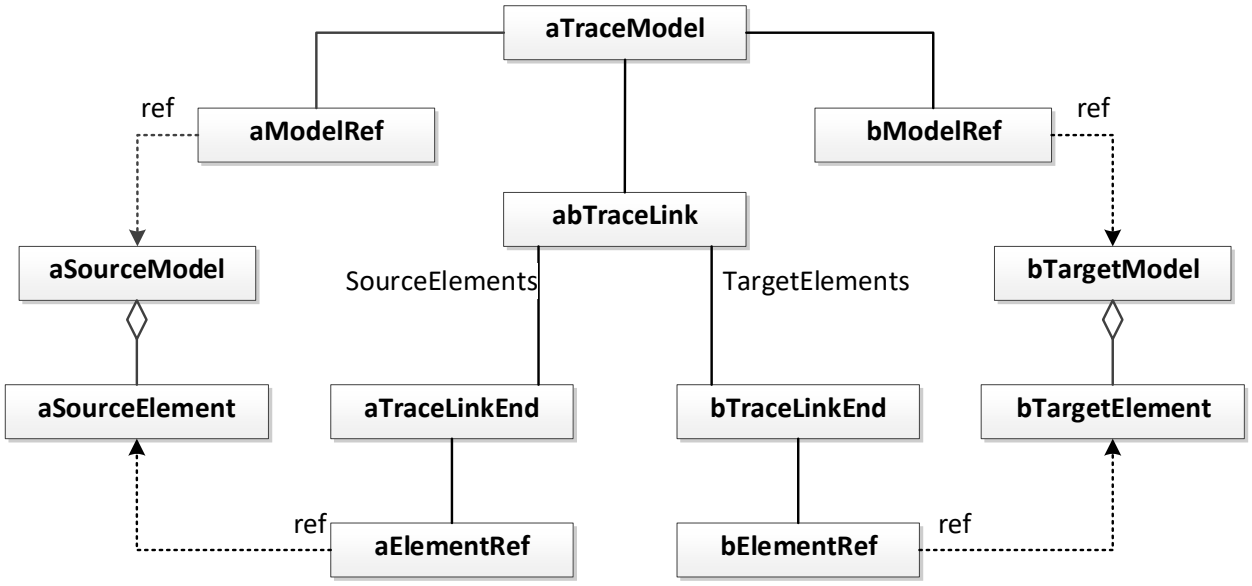


Figure 5.16: Traceability model

In the context of MDD, traceability schemes are usually explicitly expressed in metamodels, which are also usually linked to models specifying model transformations. Currently, there is no single standardized traceability metamodel. The traces among testing artifacts can be produced on-line, in which case traces are stored automatically by a tool as a by-product of the development activity. Or it can be done off-line, which means that traces are recorded automatically or manually after the actual development activity has been finished.

Using the modeling tool jUCMNav, the first step of the approach (model creation) is to create the UCM model. The feature path traversal algorithm is capable of exporting scenario models that conform to the EMF metamodel (Ecore) implementation of the UCM notations. The exported scenarios have exhaustive coverage of the UCM model and used as input to the first transformation. Implementation of the second step (model transformation) is based on the "behavioral scenarios to ATCs scenarios" model transformation. The "*ATC Builder* process" receives as input an exported scenario model (Source) and transforms it into TDL Test Configuration and Test Description models (Target). To support traceability, we enhanced the transformation tool to create traces that relate the model elements between Source and Target. Guided by a traceability scheme defined in **Table** 5-4, we recorded the produced traces in the

traceability model "*tracemodel.ecore*". Implementation of the third step test case generation and traceability information takes places when the transformed TDL specifications and the data model developed earlier become ready. We again recorded the traces, obtained as a product of the transformation, with the guidance of the traceability scheme in **Table** 5-4.

Table 5-4: Traceability scheme

Testing artifacts	What information to record	Constraints	Source
UCM Scenario	Component		Scenario Definition
	Interaction		Scenario Definition
	Action Reference		Scenario Definition
TDL Test Specification	Test Configuration		Connected components
	Test Description		Set of Interaction & Action reference
	Gate	No duplication in Gate	Component
	Interaction, Action Reference		Interaction & Action reference
	Data Instance, Data Set	No duplication in Data Set	Data model
	Port	No duplication in Port	
TTCN-3 Test case	Record, Record field		Gate
	Send, Receive		Data model, Data Set, Data Instance
	Template,		Interaction
	Function		Data model
			Action Reference

In the following section, we explain how relationships among the testing models are recorded by our developed tools in the trace model during TCG process.

5.3.2.2 Approach Realization

The LGS case study, presented earlier, is used to demonstrate the realization of the traceability approach.

I. Traceability Links Between Requirements and ATCs

During the execution of the TCG process, the UCM scenarios describing the LGS requirements are created as step 1 of the traceability approach (**Figure** 5.15). The transformation of the UCM scenarios into ATCs and the creation of traceability information take place in step 2 in the figure. Followed by step 3; transforming the ATCs into ETCs and creating the corresponding traceability information. **Table** 5-5 shows the test data extracted from the UCM "*DeploymentSucceeded*" scenario depicted in **Figure** 5.13.

Table 5-5: Extended Test data for "*DeploymentSucceeded*" Scenario

Test Data Requirement	UCM responsibility Stimulus/Response	TDL Data Instances	TTCN-3 Template
Stimulus to be sent when Pilot switches handle down	*Handle_Down*	instance Handle_Down	Template String Handle_Down_Type
Response to be received when LGCU locks doors in opened position	*LockDoorsInOpenedPos*	instance LockDoorsInOpenedPos	Template String LockDoorsInOpenedPos_Type
Response to be received when LGCU activates Gear maneuvering	*AmberON*	instance AmberON	Template String AmberON_Type
Response to be received when LGCU locks Gears in down position	*GreenON_AmberOFF*	instance GreenON_AmberOFF	Template String GreenON_AmberOFF_Type
Response to be received when LGCU locks doors in closed position	*LockDoorsInClosedPos*	instance LockDoorsInClosedPos	Template String LockDoorsInClosedPos_Type

The transformed ATC model, composed of Test Configuration, Test Description and Data Set elements is depicted in **Figure** 5.17.

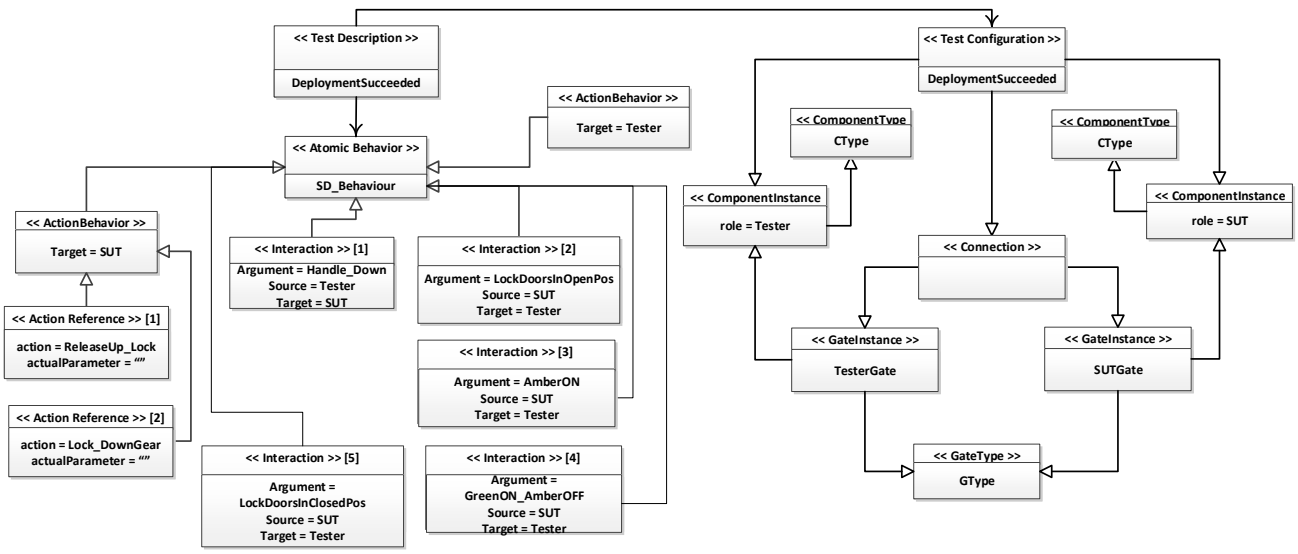


Figure 5.17: ATC model for "*DeploymentSucceeded*" scenario

Traceability information for the test configuration is depicted in **Figure** 5.18. The traceability model is named *TraceUCMModel2TDLModel*. It relates models *UCMScenarioModel* and *TDLTestScenarios*. It has one trace link named *DSScenarioTraceLink* that relates the *UCMDSScenario* in the *UCMScenarioModel* to the *TDLDSTTestSpecification* in the *TDLTestScenarios*. *DSSScenarioTraceLink* has many children; **Figure** 5.18 shows the link

DSTestConfigurationTraceLink, which relates the component Instances (Pilot and LGCU) in the *UCMDSScenario* to the gate instances (Tester and SUT) in the *TDLDSTestSpecification*.

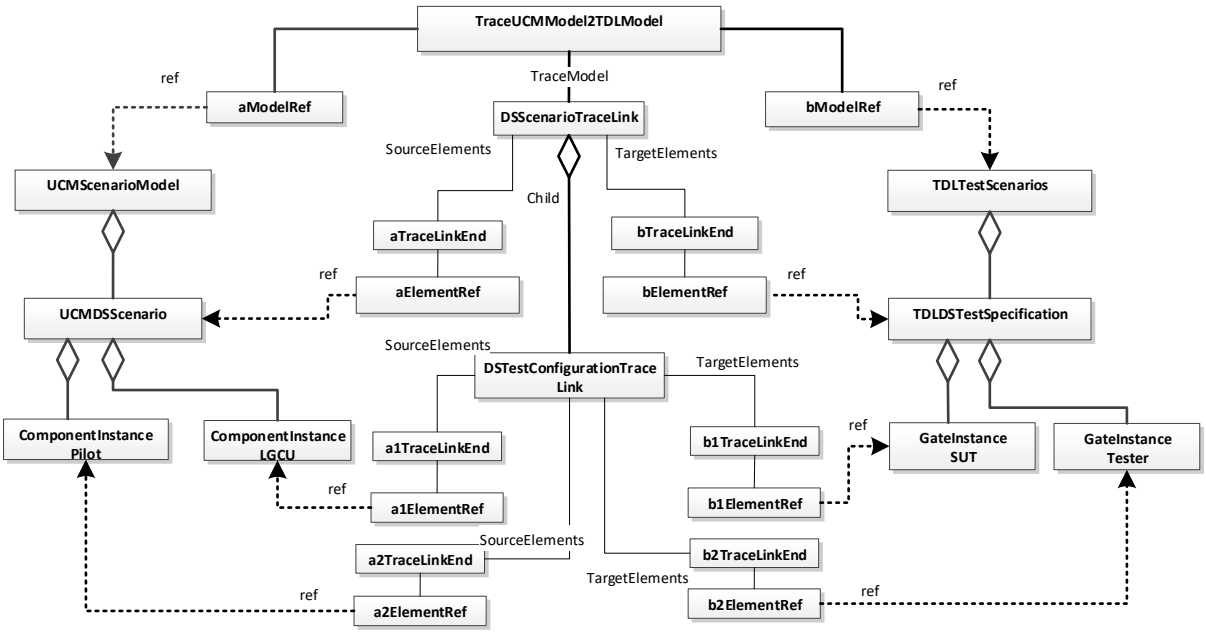


Figure 5.18: Traceability links between "*DeploymentSucceeded*" scenario and Test Configuration element.

Part of the traceability information for the test description is depicted in **Figure** 5.19. The trace link *DSSScenarioTraceLink* has another child *DSTestDescriptionTraceLink*, which relates the interactions and action references in the *UCMDSScenario* to the interactions and action references in the *TDLDSTestSpecification*. The figure shows one "Interaction" and one "Action Reference".

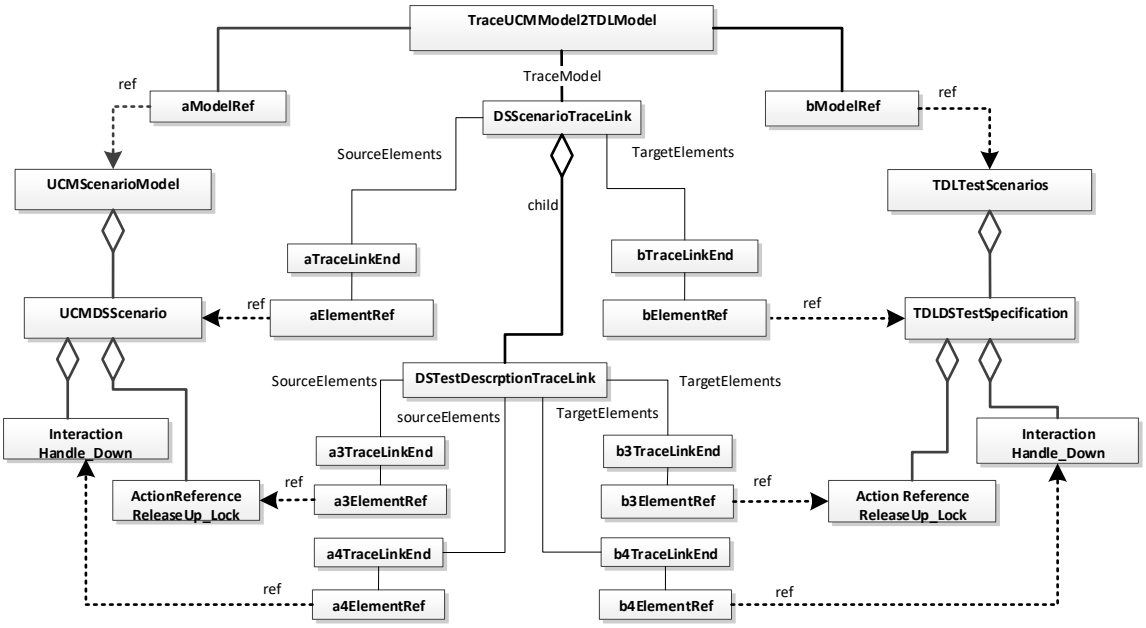


Figure 5.19: Traceability links between "*DeploymentSucceeded*" scenario and Test Description element

II. Traceability Links Between ATCs and ETCs

The last step in the approach (step 3 in **Figure** 5.15) is the generation of test cases and the creation of the traceability information among TDL test model and the generated test cases. Information from the data model in **Table** 5-5, from the TraceModel in **Figure** 5.16 and from test specification model in **Figure** 5.18 is used to complete the step. The data model is developed from the testing requirement and represents the input space for the scenario model "*DeploymentSucceeded*" under transformation. The *instances* in the data model are grouped into two sets; stimulus (Tester) and response (SUT) to build the TDL *Data Set*s element. Each *Data Set* is mapped to records and record fields (variables) in TTCN-3 syntax based on transformation rules. In **Figure** 5.20, the trace link *DSSScenarioTraceLink* has a child *DSTestDataModuleTraceLink*, which relates the Data Set, Data Instance and Interaction in the *TDLDSTestSpecification* to the Record, Record field and Send in the TC_DS_[seq]. The figure shows one "Data Set" one "Instance" and one Interaction.

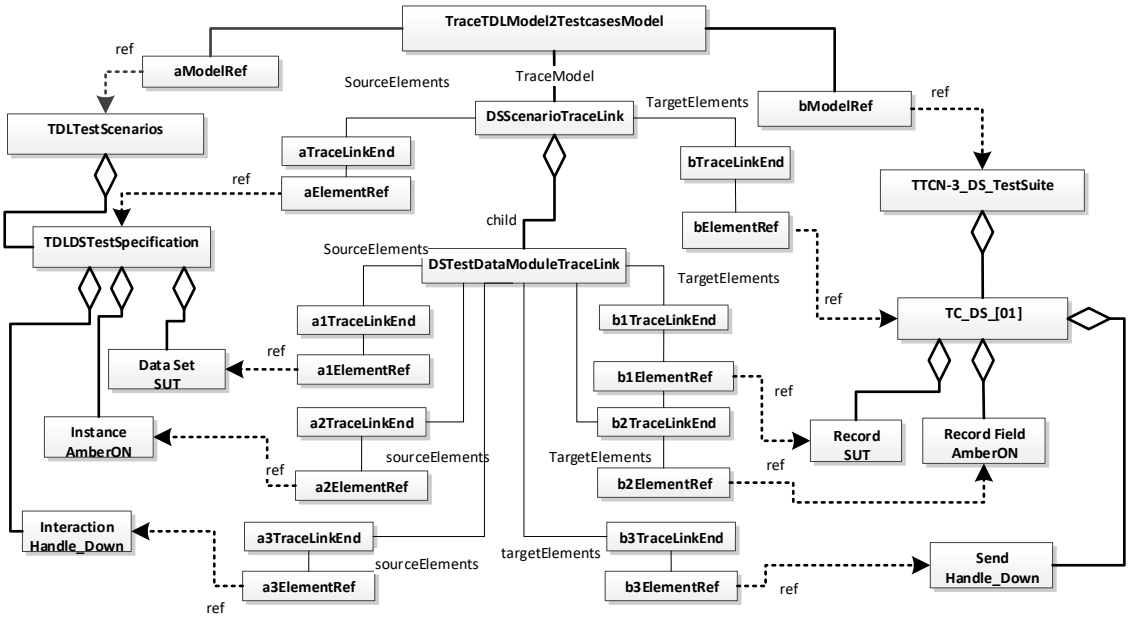


Figure 5.20: Traceability information between TDL and TTCN-3

The TDL test scenario "*DeploymentSucceeded*" is transformed into a test case in TTCN-3 by applying the structural transformation, e.g., a TDL element is transformed into a TTCN-3 module. Therefore, the resulting test case is composed of three types of modules: (1) a Test Configuration module, (2) a Test Description module, (3) and a Data module. After the TTCN-3 Data module is partially generated and test data becomes available, the module is completed with test inputs and oracle information. A new test case is added "*TC_DS_01*" to the test suite "*TTCN-3_DC_TestSuite*" for each new pair of test input and expected output found in Data model in **Table** 5-3.

III. Compliance with DO-178C Standards

In our trace model, we have trace data that shows the HLR described as TDL elements are traceable to software requirements (UCM elements) and that the LLR are traceable to HLR. The test scenarios are traced indirectly (via UCMs) to the HLRs and LLRs. The executable TCs are traced to the abstract test scenarios in TDL. Therefore, compliance with DO-178C standard is achieved for the traceability objective. LLRs are developed from HLRs, and as defined by DO-178C, an association between a requirement and its related items is necessary. The TDL can be produced from UCMs developed from HLRs or LLRs: the methodology is applicable to HLR- or LLR-based testing.

5.4. MDTGL Approach Summary

This chapter proposed a new testing methodology that automates with limited resources two major testing activities for testing ES based on modeling and model transformation. First, the chapter presented an approach for generating executable test cases from system requirements modeled with UCM notation. The TCG approach used test description language to transform the abstraction of a test description to an executable test case. The automatic development of TCs by the TCG approach has produced different models at different levels of abstraction. Next, the chapter presented a framework that aligns the activities of requirement traceability to testing to improve system quality and project cost. The traceability framework automatically links the intricate relationships among test-related artifacts, obtained as a product of the transformation, to support the automation of testing activities such as coverage measurement, result evaluation and selective regression testing. **Figure** 5.21 shows the two testing activities.

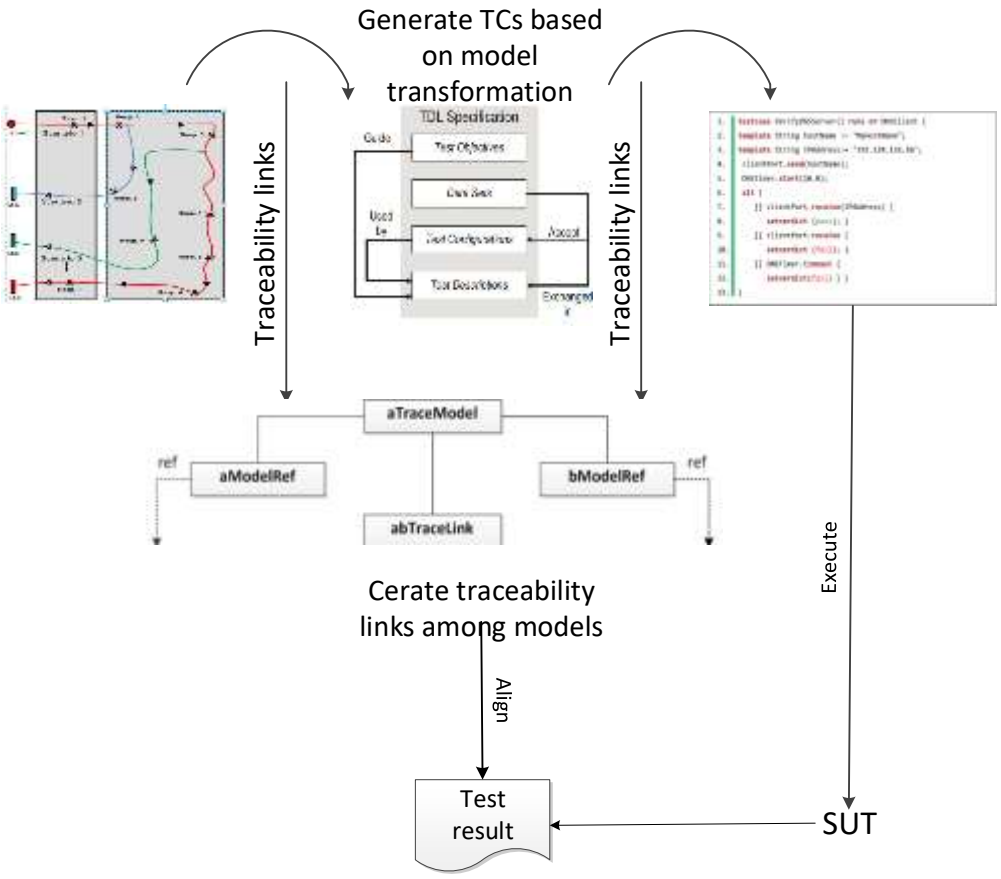


Figure 5.21: The activities of MDTGL methodology

In terms of validating the proposed testing methodology and demonstrates if it is technically feasible, the LGS case study from the avionics public domain was applied. The experiments showed that the testing artifacts are generated at the reasonable effort.

Chapter 6 TCG Approach Evaluation

6.1. Topic Overview

The evaluation of the approach is sampled with an industrial product from the private domain, an FMS, see **Figure** 6.1**.**

This activity answers the research question RQ3: "*how do we assess the correctness of a test case generation process and how to evaluate its efficiency?"*

Figure 6.1: FMS Front Panel (photo Esterline CMC Electronics)

The FMS test stimuli are key presses and the test oracles are screen dumps. Since the FMS functionality was tested using software tests developed manually and determined correctly the FMS behaviour, we wanted to evaluate our approach using the same case study to assess the efficiency of our approach. We present an empirical evaluation of the approach, based on the results obtained with 3 FMS use cases. We studied the approach efficiency in terms of generating ETCs and we evaluated the correctness of the generated workflow in two steps:

- **Perform requirement-based test coverage analysis**: we analyzed the trace model *tracemodel.ecore*, obtained as a product of the transformation, along with the generated ATCs

and ETCs to confirm that there is at least one ATC for each requirement and all ETCs and ATCs are traceable to requirements (UCM models).

- **Perform verdict analysis**: we used a set of legacy ETCs to assess the correctness of the generated ETCs. Since the execution of the legacy ETCs against SUT reported correctly its behaviour and verified that the implementation satisfies the requirements, we used them as an oracle version. We compared our ETCs verdicts against the ones emitted by the legacy ETCs. The pass verdict indicates correct implementation where the fail verdict indicates an error has been detected.

The remainder of this chapter is organized as follows. Section 6.2 presents the FMS as the case study followed by the experiment method that we used in Section 6.3. The efficiency of the approach is presented in Section 6.4. The traceability links and their alignment with testing are presented in Section 6.5. A discussion with generalization of the approach and set of lessons learned showing the difficulties encountered are presented in Section 6.6.

6.2. The Case Study FMS

An FMS is typically comprised of the following interrelated functions: navigation, flight planning, trajectory prediction, performance computations, and guidance. It provides the primary navigation, flight planning, optimized route determination and en-route guidance for an aircraft. To accomplish these functions, the flight management system must interface with several other avionics systems. A short description of three key functions performed by the FMS and used in the evaluation is given below:

- **Flight Planning**: the flight planning function allows the creation of a flight plan based on the data combinations from a company's route, defined waypoints, navigation database, etc.
- **Lateral Guidance**: This function allows waypoint management via its control display unit interface when an aircraft is configured as a rotor.
- **Navigation**: This function determines the accuracy variable based on the present position, ground speed, and wind speed/wind direction.

6.3. The Experimental Method

We analyzed the efficiency of the approach by running an experiment aiming to determine whether the approach is efficient to generate ATCs that cover the requirements and can be

transformed to correct ETCs. We consider an ETC is correct, after being executed on the FMS if it reports correctly the behaviour of the SUT. Our first step was to select from the legacy software tests a number of ETCs that cover the three FMS key functions reported in the previous section. Five legacy ETCs that were manually developed, performed on the FMS and reported correctly its behaviour covered those functions and therefore were selected. Next, we identified the corresponding requirements of these legacy ETCs and grouped them into 3 use cases. The description of each use case is given as follows:

— **Automatic Leg Transitions:** contains 8 functional requirements that specify the automatic leg change using fly-by (turn anticipation) or fly-over (turn over the waypoint).

— **Provide Guidance for a Manual Direct-to Intercept:** contains 7 requirements that specify the operations of the "discontinuity ahead" alter message on the modified route.

— **Predict the Expected Time of Arrival (ETA) with different configurations:** contains 9 requirements that specify the computations to be performed by the FMS for an aircraft to arrive at a certain place.

For each use case, the experimental method we applied consists of:

- Requirement stage: the requirements in the use case were formalized into Cockburn notation and manually mapped to UCM models. We validated the scenario models and checked if they describe correctly all the requirements.
- Test scenario stage: for each possible path in the scenario model, its definition was created and stored as an XML file. Using our java-based tool, we transformed the scenario path expressed in XMI format into scenario test expressed in TDL notation. We completed the obtained ATC with *Test Objectives* and *Data Instances* elements which are taken mainly from the requirements.
- Test generation stage: based on the transformation tool that we implemented with the *Xtext* and *Xtend* framework, we transformed each ATC into an executable ETC.
- Test execution stage: the resulting ETCs that correspond to the selected legacy ETCs were executed on the FMS and their test results were recorded.

As a result, 26 ATCs and ETCs were generated from the 3 use cases. Five ETCs were performed on the FMS and their test results were recorded. The selected ETCs stimulate the FMS functionality and reflect largely the use cases. **Table** 6-1 shows the details about the executed

ETCs where the description of each ETC is given in column 1. Columns 2 and 3 show the number of exchanged messages with the FMS and their verdict respectively. A total of 803 exchanged messages and 338 test verdicts are performed as shown in the total row.

Table 6-1: The executed TPs against the FMS

TP performed on FMS	# of input/output exchanged with FMS	# of verdict per TP
Fly-by procedure	32	10
Fly-over procedure	26	11
Fly-over procedure via DES+SAR	236	129
Manual Direct-to Intercept	116	20
ETA Computation	393	168
Total	**803**	**338**

6.4. Requirement Coverage and Generating Correct ETCs

We analyzed the generated ATCs to check if they cover the requirements. **Table** 6-2 shows that the approach covered all paths in the scenario models effectively. In fact, the approach generated one ATC for each scenario path in the scenario model. The total number of the generated ATCs successfully covers all possible paths in the UCM model and achieves therefore full scenario and requirement coverage.

Table 6-2: The requirement coverage by the generated ATCs from UCM model

Use case modeled as scenario	# of Scenario Path		# of ATCs	Requirement Coverage Rate
	Main	Secondary		
Automatic leg transitions	3	9	12	100 %
Provide Guidance for a Manual Direct-to Intercept	1	7	8	100 %
Expected Time Arrival Computation	1	5	6	100 %

The generated ETCs were assessed for their correctness by comparing their test results against the legacy ETCs. The objective is to have the ETCs behaviour matches the legacy tests. As mentioned, the legacy tests are used as a golden version to assess the correctness of the generated TPs. **Table** 6-3 shows the result of the verdict comparison for each pair of ETC. The scenario models that describe the requirements are shown in the first column. Followed by ETC description in the second column. The rate of matching verdict with the corresponding legacy test is presented in the third column.

Table 6-3: The matching rate of the executed ETCs

Use case modeled as scenario	Executed ETC	Verdict matching rate with legacy
Automatic Leg Transmission	Fly-by procedure	100 %
	Fly-over procedure	100 %
	Fly-over procedure via DES+SAR	98 %
Provide Guidance for a Manual Direct-to Intercept	Manual Direct-to Intercept	97 %
Expected Time Arrival Computation	ETA Computation	98 %

All the verdicts in the *Fly-by-procedure* and *Fly-over-procedure* ETCs matched the corresponding verdicts of the legacy tests. In the remaining ETCs, *Fly-over-procedure via DES+SAR, Manual Direct to-Intercept* and *ETA computation*, very few numbers of verdicts did not match with the corresponding legacy tests. The result in the third column determined with a high rate of success the SUT behaviour— emitting pass verdict when it is expected and fail verdict in the presence of errors.

6.5. Traceability Links and Alignment with ETCs Result

The result of the test case generation process in the previous section is the trace model "*tracemodel.ecore*" which relates UCM scenario models to TTCN-3 test cases grouped in test suites. Each test case, generated within a unique identifier, is a sequence of actions and

interactions with defined input parameter values and output parameter values. The execution of the test case results in the assignment of a test verdict; pass or fail. In the "*tracemodel.ecore*", the links between requirements and ETCs may have several possible cardinalities:

- One-to-one: one requirement is tested exactly by one ETC and this test case tests only this requirement.

- One-to-many: one requirement is tested by several ETCs and these ETCs participate to test only this requirement.

- Many-to-many: one requirement is tested by several ETCs, which are used to test several requirements.

Figure 6.2 shows the relationships between the testing artifacts for the "*DeploymentSucceeded*" scenario. The traceability link *DSScenarioTraceLink[1]* relates the model *UCMDSScenario* to the model *TDLDSTestSpecification* which is related to several test cases via the traceability link *DSScenarioTraceLink[2]*. The generated test cases are children of the test suite *TTCN-3_DS_TestSuite*.

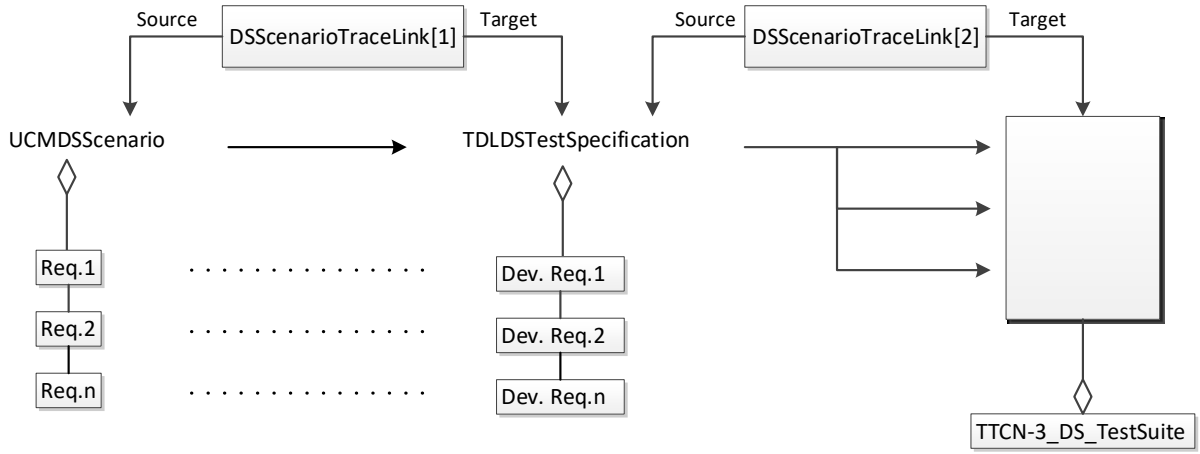


Figure 6.2: Requirement Traceability among testing models

The trace model takes a significant importance in the TCG process. On one hand, it provides a clear meaning for each generated ETC: the tested requirement(s) gives the purpose of the associated test case(s). It is a kind of rationale for the generated test suite. On the other hand, the trace model exhibits clearly which requirements are actually tested (and how), and which requirements are not tested. For the not tested requirements, this suggests completing the test

suite to obtain full functional coverage. During the test execution of the ETC in Listing 6-1, the traceability links in the trace model help to identify the related requirements when it fails. Similarly, when the test case passes, they certify that the related requirements were implemented and tested.

```
1.  module TestDescription {
2.    import from TestConfiguration all; import from TestData all;
3.    testcase TC_DS_01 () runs on Pilot {
4.      map (mtc:gPilot, system:gLGCU);
5.      timer deploymentTime; timer lockDoorOpenedTime;
6.      timer gearsManeouvringTime;  timer gearLockedDownTime;
7.      timer lockDoorClosedTime;
8.      Handle_Down();  // function call
9.      gPilot.send(GearDownTemplate);
10.     deploymentTime.start(15.0);
11.     OpenDoors();  // function call
12.     LockDoorsInOpenedPos ();
13.     lockDoorOpenedTime.start(7.0);
14.     alt {
15.       [] gPilot.receive(LockOpenedDoorTemplate) { setverdict(pass);
16.          ReleaseUp_Lock();
17.          gearsManeouvringTime.start(7.0);}
18.       [] lockDoorOpenedTime.timeout { setverdict(fail) }
19.       [] gPilot.receive(AmberLightTemplate) { setverdict(pass);
20.          Lock_DownGears();
21.          gearLockedDownTime.start(7.0); }
22.       [] gearsManeouvringTime.timeout { setverdict(fail) }
23.       [] gPilot.receive(GreenLightTemplate) { setverdict(pass);
24.          CloseDoors();
25.          LockDoorsInClosedPos();
26.          lockDoorClosedTime.start(7.0);
27.       [] gearLockedDownTime.timeout {  setverdict(fail) }
28.       [] gPilot.receive(LockClosedDoorTemplate) { setverdict(pass);
29.          ConfirmGearsDown(); }
30.       [] lockDoorClosedTime.timeout { setverdict(fail) }
31.       [] deploymentTime.timeout { setverdict(fail) } }
32.     unmap (mtc:gPilot, system:gLGCU); }  }  }
```

Listing 6-1: ETC TTCN-3 generated from “DeploymentSucceded” scenario

6.6. Discussion of TCG Approach

We applied our approach to industrial case study FMS at the Avionic industry. The validation has been achieved by comparing the behaviour of the legacy and the generated tests. If they are behaviour equivalent, the same sequence of test events and verdicts, we can consider them comparable. The verdict of almost all oracle steps in the generated ETC matched their corresponding ones in the legacy. In other words, the generated ETCs passed and failed in the same steps as the legacy ETCs did except a small number of failures in the generated tests. These failures were mostly due to timing issues. The generated tests in TTCN-3 execution have a considerably better performance as the legacy system and the SUT is relatively slow. These cases could be easily detected using the state of the SUT. If the state was the same as for the preceding test event, this indicates that the SUT has not updated its state yet. Here, the responses are not coming spontaneously but instead, the test system must query the SUT to obtain the response. Also, some of the failures could indicate that there are alternative behaviour in the SUT, something that the legacy test system could not handle because it was based on linear sequences of test events.

In conclusion, this study reveals that our approach generated ATCs that cover all the described requirements in the scenario models achieving full requirement coverage.

Compared to the legacy testing system, the new approach improves the testing in practice and offers several advantages to the test engineers. We found the following benefits from our new testing practice:

— **Increased test system understanding:** using a model enables to get an overview of the behaviour of a system compared to scattered bits and pieces of information.
— **Early Testing:** The test engineers don’t need to wait; they describe the requirements in a model and then push a button to generate the tests.
— **Reduced test effort**: in our model-driven testing, the number of iterations to get correct ETCs is reduced. The test development phase is eliminated. The ETCs are no longer written by hand or manually corrected, but generated.
— **Traceability:** Traceability links among testing artifacts are generated during model transformation. Since ETCs are derived from the UCM models where requirements are

described, any defect found during the execution of an ETC can be traced back to its requirement.

— **Systematic and automation:** with the help of the developed tools, repeated tests are enabled which ensures the robustness of the test results.

— **Reduced human errors**: The fact that the tests are generated from the model and thus consistent with requirements reduces, by definition, the possibility of error in the test suite.

6.6.1 Generalization of the Approach

The approach focuses on functional aspects of software and has been applied to two realistic case studies from the avionics domain. Additionally, the methodology can apply to safety-critical software as it covers timing requirements and provides traceability evidence from requirements to tests. The approach relies on two major elements to improve the testing process:

Modeling: the system requirements (functional) and design are described by high-level visual models and DSL abstracting away technological implementation detail.

Model transformation: the automated model transformations are used to generate tests to reduce the manual work, to provide traceability evidence and to simulate high-level models to validate the suitability of the modeled system behaviour in an early development phase.

Today, the practical realisation of model-driven testing benefits from a variety of tools and technologies. Some requirements may not be describable with the UCM notation such as robustness requirements. Such requirements have to be specified through other notations or languages. The model transformations are (partially) automated and require little human intervention. The process converts the informal requirements into a formal UCM model. We have used the tool described in [17] that generates individual test traces, called test scenarios in TDL but as already mentioned, test traces are not always test cases. A good test case comprises alternative behaviour both in TDL and in TTCN-3. This part is post-processed with a tool to resolve the absence of alternatives in the scenario metamodels targeted by jUCMNav's traversal mechanism. The hints found in [17] have been tried out and were successful. However, the translation from TDL to TTCN-3 is relatively straightforward since there is mostly a one to one mapping from TDL to TTCN-3. Only, things such as describing test purposes are not covered and thus have to be translated manually usually as TTCN-3 comments. Overall, our achievement was to show that it is an advantage to build a formal UCM model because everything else down

the path can be automatically generated and is either all the way right or all the way wrong. Test automation has the advantage to be systematic when it comes to errors as opposed to manual processes where errors are introduced randomly and are difficult to trace. This automation reduces the required amount of manual work for test development, such that the testing process is supposed to become less error-prone and more efficient.

6.6.2 Lessons Learned

We distill some of the important lessons we have learned in developing and deploying the testing methodology.

The users of the testing methodology should not need to have the functional requirements expressed with use case notation to model them as scenarios. However, requirements presented as a use case facilitated the mapping to UCM models. The model transformation to TDL domain is not fully automatic and requires human intervention to obtain the data elements and to construct the alternatives. The TDL models were a key component of model-driven testing as they have been used as input and output in the model transformation process. The decision to use the TDL notation in the development of tests was successful. TDL narrowed the gap between the described requirements and tests and served as a way of communication with non-technical people and as a base to generate concrete tests.

Chapter 7 Conclusions

7.1. Topic Overview

ESs have increasing importance in modern society due to the close interaction with their environment. Ensuring high-quality software that is of crucial importance today is often costly. Quality assurance efforts, especially testing efforts, often consume more than 50 % of the overall development efforts [138], [139]. Therefore, testing an ES implementation with limited resources to ensure that it is fault-free before its deployment is necessary. Several new technologies have emerged to address the growing demand for ES software verification. One of those techniques is MDT which is an automation of MBT that uses model-transformation technology on formal models, their meta-models, and transformation rules defined in terms of mappings between the elements of meta-models.

While many researchers have found methods of improving UCM-based testing by deriving test goals, its abstraction level remains inappropriate for the generation of implementation-level test cases. Moreover, UCM models abstract from detailed communication mechanisms, and emphasize behavior rather than data which makes deriving executable test cases a difficult activity. There are a number of important related issues that need to be researched such as generating test cases from UCM scenarios with limited resources. Furthermore, there is little research done on linking the activities of requirement traceability with testing. As a result, it is important to develop a valid and flexible approach that can handle these issues.

In this chapter, Section 7.2 summarises the research findings of each chapter. Section 7.3 explains how research objectives are achieved. A summary of the Thesis contributions is then presented in Section 7.4. Finally, Section 7.5 identifies the research limitations and points to future research ideas.

7.2. Research Summary

The aim of the research presented in this Thesis was to develop, validate and automate a flexible model-driven testing approach based on modeling and model transformation for testing ESs.

Chapter 1 gave an overview of the area under research and highlighted the motivation of this research. That emphasized the need for developing a valid model-driven testing approach capable of testing ESs with limited resources. A set of research objectives were identified to fulfill the research aim followed by Thesis contributions.

Chapter 2 reviewed the related literature that addressed testing ESs. The concept of testing was defined and explained by addressing some topics related to testing types. Several studies were reviewed in this chapter that covers three testing activities; (1) model transformation, (2) test case generation, and (3) requirement traceability and alignment with testing.

Chapter 3 introduced the three domain-specific languages UCM, TDL and TTCN-3 where their metamodels are used in requirement propagation and model transformation. The construct of each language is described extensively with example.

Chapter 4 presented a reverse engineering process aiming to discover a path from TDL to TTCN-3. The process reversed engineer a legacy software test by migrating test cases written as Ant/xml files into the TTCN-3 code. The obtained executable test cases are re-engineered to a higher level of abstraction to obtain abstract test cases in TDL notation.

Chapter 5 developed the MDTGL methodology based on modeling and model transformation that automated the generation of test cases, the traceability requirement among testing artifact, and the checking result of interaction behavior. Several tools have been developed that target the automatic testing of ESs. The validity of the MDTGL was empirically demonstrated by running it on a public case study.

Chapter 6 assessed and evaluated the new testing approach based on assessment factors which considered requirement coverage, the correctness of generated workflow and labor cost with respect to the length of generated test cases. The chapter presents an experiment applied to the avionics case study for estimating the assessment criterion. A discussion with generalization of the approach and set of lessons learned showing the difficulties encountered especially for testing ES is then highlighted.

7.3. Meeting the Research Objectives

The main aim of the Thesis was to provide software engineering community with a sound, valid and flexible testing approach for testing ESs with limited resources. This section shows how this research successfully achieved its objectives.

Objective 1: *"To determine the differences and obstacles that reside among the three languages; UCM, TDL and TTCN-3"*. The first objective was achieved in Chapter 2 and 3 by studying the constructs of each language and its metamodel.

Objective 2: *"To resolve the obstacles and differences that exist among the three languages and demonstrate the approach feasibility"*. The second objective was achieved in Chapter 4 and 5 by developing transformation rules between the three languages and discovering a path from UCM scenarios to TTCN-3 test cases via TDL.

Objective 3: *"To generate test cases in TTCN-3 from UCM models via TDL based on requirement analysis, model transformation and refinement process"*. The third objective was achieved in Chapter 5 by developing a test case generation approach based on model-driven technique to derive testing artifacts. Next, by developing a TCG process for generating executable test cases. The technique can be seen as a process of successive refinements of specifications that involves model transformation and the insertion of additional information.

Objective 4: *"To align traceability requirement with generated test artifacts and testing"*. This objective was achieved in Chapter 5 by extending the MDTGL tool to create explicit relationships in a trace model among generated testing artifacts.

Objective 5: *"To validate the generated testing artifacts in terms of effectiveness and usefulness at the specification and implementation level"*. This objective was achieved in Chapter 6 by sampling the new approach with an industrial ES and compared it to the testing approach.

Objective 6: *"To develop and provide traceability evidence from requirements to tests for compliance with DO-178C standards"*. This objective was achieved in Chapter 4 by developing a framework that creates traceability links in recorded them in a trace model.

7.4. Summary of Research Contributions

The main research contributions are summarized in the following subsections.

7.4.1 Towards Building Model-Driven Testing Methodology

The reengineering of legacy software tests aims to discover feasible transformation from the test layer to test requirement. Furthermore, it is used to help build the model transformation, generate TTCN-3 test cases from TDL models, and show its feasibility. Then, after showing that TTCN-3 test cases can be derived from TDL models, the approach is extended with the requirement layer which describes software specifications in UCM scenarios where test objectives can be driven and transformed into TDL models. Reaching this point, the feasibility of transforming TTCN-3 scripts into a TDL model is shown, and a forward engineering process to regenerate the test cases can be undertaken.

7.4.2 Test Case Generation Approach

Several model-based testing approaches have been proposed to improve UCM-based testing by deriving test goals. However, most of these approaches stopped at the generation of abstract test cases. Another challenge besides transforming UCM scenario models to test cases in a scripting language is the validation of the transformation, both in terms of technical correctness and usefulness.

Other research assumed that there are unlimited resources to generate the testing artifacts. It is thus essential to consider an approach that generates with limited resources executable test cases and validating them in the industrial case study. The lack of a mature test case generation process based on UCM models directed our research to develop one.

The concept of test case generation was proposed to support the testing of ES with limited resources. As a result, we focused on generating test cases. We developed models to describe the system requirements and rules to transform them up to test cases.

7.4.3 Requirement Traceability and Alignment with Testing

The alignment research area model-based development has attracted a lot of attention. The idea behind MBT is the derivation of executable test code from test models by analogy to MDA [140].

One challenge in using MBT approach for aligning requirements and testing is to make test cases executable, as the tests are not at the same level of detail as the implementation code [141]. This technique is becoming of more interest in industry because it provides automatic deriving of test cases from the behavioral model of the system called the test model. Our traceability model, obtained as a product of model transformation during the TCG process, helped determine what requirement has been covered by which test and how the generated ETCs cover these requirements. Another important reason for traceability is improving change management by helping to find out how a change in the requirement is reflected in the ETCs. It also helped trace from tests back to requirements which is helpful to find the root of a failed test. Furthermore, compliance with DO-178C standard is achieved for the traceability objective.

7.4.4 The Application of TCG Approach on an Industrial Case Study

Some proposed approaches in the literature lack automation tool support. Using such approaches requires a deep understanding of their mechanism and significant manual effort in generating and executing test cases. Others were partially automated. Their tools were responsible for only automating the generation of test input which requires other sets of tools to make the test cases executable.

To our knowledge, there has yet to be a study that compares the efficiency of similar approaches on real applications. This research used an industrial ES with well-identified assessment criteria by which the efficiency of testing approaches can be compared were also presented. In summary, we aimed to develop a testing approach capable of detecting as many faults as possible with limited resources. The study at the implementation level confirmed results obtained at the specification level. Our TCG approach reduced the test effort and allowed to start testing early.

7.5. Research Limitations and Future Work

This section identifies a set of research limitations encountered and suggests a set of complementary future work to address them.

7.5.1 Case Studies

This research succeeded in comparing the efficiency of MDTGL with industrial testing approach based on specification case studies. However, the relatively small size of case studies used can be

considered a limitation. Choosing small specification models for the approach application was justified due to the limited access imposed by our research partner.

For future research, we may use more industrial case studies by which more functional faults can be found and categorized. Moreover, comparing the results with MDTGL.

7.5.2 Automation of Recording Traceability Links

The goal of creating traceability relations among testing artifacts during the development of TCG process is achieved. However, the recording of these traceability links in our trace model is not automated. Therefore, there is an automation direction for future work to automate the process of recording traceability links in the trace model to ensure the benefits of maintaining traceability relations over time as the software system evolves.

References

[1] M. Zhang, T. Yue, S. Ali, H. Zhang, and J. Wu. A systematic approach to automatically derive test cases from use cases specified in restricted natural languages. In Proceedings of the 8th International Conference on System Analysis and Modeling: Models and Reusability (SAM'14), 2014.

[2] Elberzhager, F., Rosbach, A., Münch, J., & Eschbach, R. (2012). Reducing test effort: A systematic mapping study on existing approaches. Information and Software Technology, 54(10), 1092-1106.

[3] Grieskamp, W., Kicillof, N., Stobie, K. and Braberman, V. (2011) Model-based quality assurance of protocol documentation: tools and methodology. Software Testing, Verification and Reliability, 21 (1), pp. 55-71

[4] What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry? Manfred Broy (Technical University Munich, Germany), Sascha Kirstan (Altran Technologies, Germany), Helmut Krcmar (Technical University Munich, Germany) and Bernhard Schätz (Technical University Munich, Germany) DOI: 10.4018/978-1-61350-438-3.ch013.

[5] Baker, P., Dai, Z. R., Grabowski, J., Schieferdecker, I., & Williams, C. (2007). Model-driven testing: Using the UML testing profile. Springer Science & Business Media. ISBN 9783540725626.

[6] Baudry, Benoit, et al. "Barriers to systematic model transformation testing." Communications of the ACM 53.6 (2010): 139-143.

[7] Boucher, Mathieu, and Gunter Mussbacher. "Transforming workflow models into automated end-to-end acceptance test cases." 2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE). IEEE, 2017.Bruel, Jean-Michel, et al. "Model Transformation Reuse Across Metamodels." International Conference on Theory and Practice of Model Transformations. Springer, Cham, 2018.

[8] He, C., & Mussbacher, G. (2016, September). Model-driven engineering and elicitation techniques: a systematic literature review. In 2016 IEEE 24th International Requirements Engineering Conference Workshops (REW) (pp. 180-189). IEEE.

[9] Buhr, Raymond JA. "Use case maps as architectural entities for complex systems."Software Engineering, IEEE Transactions on 24.12 (1998): 1131-1155.

[10] ITU-T Z.151: http://www.itu.int/rec/T-REC-Z.151/en.

[11] Duran, M.B. and Mussbacher, G., 2019. Reusability in goal modeling: A systematic literature review. Information and Software Technology.

[12] Amyot, D., Echihabi, A., and He, Y. (2004), UCMEXPORTER: Supporting Scenario Transformations from Use Case Maps. Proc. of NOTERE'04, Saïdia, Morocco, June.

[13] ITU-T – International Telecommunications Union (2002), Recommendation Z.100 (08/02): Specification and description language (SDL). Geneva, Switzerland.

[14] ITU-T – International Telecommunications Union (2003), Recommendation Z. 140 (04/03): Testing and Test Control Notation version 3 (TTCN-3): Core language. Geneva, Switzerland.

[15] OMG – Object Management Group (2003), Unified Modeling Language Specification, Version 1.5. http://www.omg.org/uml/

[16] Telelogic AB (2004), Tau SDL Suite, http://www.telelogic.com/products/tau/sdl/index.cfm

[17] Boulet, P., Amyot, D., Stepien, B.: Towards the generation of tests in the test description language from use case map models. In: SDL 2015: Model-Driven Engineering for Smart Cities, pp. 193–201. Springer (2015)

[18] http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome.

[19] Lago, P., Muccini, H., van Vliet, H.: A scoped approach to traceability management. J. Syst. Softw. 82(1), 168–182 (2009)

[20] DO-178C, available from RTCA at www.rtca.org.

[21] http://www.ttcn-3.org/index.php/downloads/standards.

[22] http://www.etsi.org/deliver/etsi_es/203100_203199/20311901/01.03.01_60/es_20311901v010301p.pdf.

[23] En-Nouaary, A., Dssouli, R., Khendek, F. and Elqortobi, A. (1998) Timed test cases generation based on state characterization technique. Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, pp. 220-229.

[24] Broekman, B. and Notenboom, E. (2003) Testing Embedded Software. London, UK: Addison-Wesley.

[25] Hessel, A., Larsen, K. G., Mikucionis, M., Nielsen, B., Pettersson, P. and Skou, A. (2008) Testing Real-Time Systems Using UPPAAL. Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, Berlin Heidelberg, pp. 77–117.

[26] Rollet, A. (2003) Testing robustness of real-time embedded systems. In Proceedings of Workshop On Testing Real-Time and Embedded Systems (WTRTES), Satellite Workshop of FM 2003 Symposium, Pisa, Italy.

[27] Mandrioli, D., Morasca, S. and Morzenti, A. (1995) Generating test cases for real-time systems from logic specifications. ACM Transactions on Computer Systems, 13 (4), pp. 365-398.

[28] En-Nouaary, A. (2008) A scalable method for testing real-time systems. Software Quality Control, 16 (1), pp. 3-22.

[29] Utting, M. and Legeard, B. (2007) Practical model-based testing: a tools approach San Francisco: Elsevier.

[30] Sugeta, T., Maldonado, J. and Wong, W. (2004) Mutation Testing Applied to Validate SDL Specifications. Springer Berlin / Heidelberg.

[31] Abou Trab, Mohammad. Software engineering: Testing real-time embedded systems using timed automata-based approaches. Diss. Brunel University, School of Information Systems, Computing and Mathematics, 2012.

[32] Briones, L. B. (2007) Theories for model-based testing: real-time and coverage. Thesis, Centre for Telematics and Information Technology

[33] Gross, H.: Testing and the uml – a perfect fit. Technical report, Fraunhofer IESE Report 110.03E (2003)

[34] Schieferdecker, I., Din, G.: A meta-model for ttcn-3. 1st International Workshop on Integration of Testing Methodologies (ITM 2004) (2004)

[35] Utting, M. (2005). Model-Based Testing. In Proceedings of the Workshop on Verified Software: Theory, Tools, and Experiments VSTTE 2005.

[36] Kamga, J., Herrmann, J., and Joshi, P. Deliverable (2007). D-MINT automotive case study-Daimler, Deliverable 1.1, Deployment of model-based technologies to industrial testing, ITEA2 Project, Germany.

[37] S. Dalal et al. (1999), “Model-based testing in practice”, In: ICSE’99, May, pp. 285—294

[38] G.M. Lima, G.H. Travassos (2005), “A Strategy for Object Oriented Software Integration Testing”. In: LATW’2005.

[39] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan and J. Kazmeier (2006), “Automation of GUI testing using a model driven approach”, In: AST'06, ACM Press

[40] J. Edvardsson (1999), “A survey on automatic test data generation”. In: 2nd ECSEL, pages 21--28. October.

[41] A. Hartman (2002), “Model Based Test Generation Survey”, Technical Report, available on 11/2006 at http://www.agedis.de/downloads.shtml.

[42] M. Prasanna et al. (2005), “Survey on Automatic Test Case Generation”, Academic Open Internet Journal, available at http://www.acadjournal.com/2005/v15/part6/p4/.

[43] B. Kitchenham (2004), “Procedures for Performing Systematic Review”, Joint Technical Report Software Engineering Group, Department of Computer Science Keele University, UK, and Empirical Software Engineering, National ICT Australia Ltd.

[44] Kienzle, Jörg, et al. "A unifying framework for homogeneous model composition." Software & Systems Modeling 18.5 (2019): 3005-3023.

[45] Jamda: The Java Model Driven Architecture 0.2, May 2003, http://sourceforge.net/projects/jamda/

[46] Mens, Tom, and Pieter Van Gorp. "A taxonomy of model transformation." Electronic Notes in Theoretical Computer Science 152 (2006): 125-142.

[47] Czarnecki, Krzysztof, and Simon Helsen. "Classification of model transformation approaches." Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture. Vol. 45. No. 3. 2003.

[48] Sendall, Shane, and Wojtek Kozaczynski. "Model transformation: The heart and soul of model-driven software development." IEEE software 20.5 (2003): 42-45.

[49] D. Milicev, “Domain Mapping Using Extended UML Object Diagrams,” IEEE Software, vol. 19, no. 2, Mar./Apr. 2002, pp. 90–97.

[50] J. Whittle, “Transformations and Software Modeling Language: Automating Transformations in UML,” Proc. UML 2002, LNCS 2,460, Springer-Verlag, 2002, pp. 227–242.

[51] OMG, The Object Constraint Language Specification 2.0, OMG Document: ad/03-01-07

[52] Rational XDE, http://www.rational.com/products/xde.

[53] Java Metadata Interface 1.0, July 2002, http://java.sun.com/products/jmi.

[54] OMG, Meta Object Facility 1.4, OMG Document: formal/02-04-03.

[55] Object Management Group, MOF 2.0 Query / Views / Transformations RFP, OMG Document: ad/2002-04-10, revised on April 24, 2002.

[56] Object Management Group. Action Semantics for the UML, 2001. ad/2001-08-04.

[57] CBOP, DSTC, and IBM. MOF Query/Views/Transformations, Revised Submission. OMG Document: ad/03-08-03.

[58] Strategies for Program Transformation, http://www.stratego-language.org.

[59] OptimalJ 3.0, User's Guide, http://www.compuware.com/products/optimalj

[60] Alcatel, Softeam, Thales, TNI-Valiosys, Codagen Corporation, et al. MOF Query/Views/Transformations, Revised Submission. OMG Document: ad/03-08-05.

[61] Interactive Objects and Project Technology, MOFQuery/Views/Transformations, Revised Submission. OMG Document: ad/03-08-11, ad/03-08-12, ad/03-08-13

[62] QVT-Partners. MOF Query/Views/Transformations, Revised Submission. OMG Document: ad/2003-08-08

[63] D. H. Akehurst, S.Kent. A Relational Approach to Defining Transformations in a Metamodel. In J.-M. Jézéquel, H. Hussmann, S. Cook (Eds.): UML 2002 - The Unified Modeling Language 5th International Conference, Dresden, Germany, September 30 - October 4, 2002. Proceedings, LNCS 2460, 243-258, 2002.

[64] Compuware Corporation and Sun Microsystems, MOF Query/Views/Transformations, Revised Submission. OMG Document: ad/03-08-07.

[65] Amyot, D. (2001), Specification and Validation of Telecommunications Systems with Use Case Maps and LOTOS. Ph.D. thesis, SITE, University of Ottawa, Canada, September. http://www.usecasemaps.org/pub/da_phd.pdf.

[66] Miga, A., Amyot, D., Bordeleau, F., Cameron, C. and Woodside, M. (2001), Deriving Message Sequence Charts from Use Case Maps Scenario Specifications. Tenth SDL Forum (SDL'01), Copenhagen, Denmark, June. LNCS 2078, Springer, 268-287

[67] Amyot, D., Cho, D.Y., He, X., and He, Y. (2003), Generating Scenarios from Use Case Map Specifications. Third International Conference on Quality Software (QSIC'03), Dallas, USA, November. http://www.usecasemaps.org/pub/QSIC03.pdf.

[68] Amyot, D., Echihabi, A., and He, Y. (2004), UCMEXPORTER: Supporting Scenario Transformations from Use Case Maps. Proc. of NOTERE'04, Saïdia, Morocco, June.

[69] UCM User Group (2003), UCMEXPORTER, http://ucmexporter.sourceforge.net/

[70] ITU-T – International Telecommunications Union (2004), Recommendation Z.120 (04/04): Message sequence chart (MSC). Geneva, Switzerland

[71] OMG – Object Management Group (2003), Unified Modeling Language Specification, Version 1.5. http://www.omg.org/uml/

[72] He, Y., Amyot, D., and Williams, A. (2003), Synthesizing SDL from Use Case Maps: An Experiment. 11th SDL Forum (SDL'03), Stuttgart, Germany, July. LNCS 2708, Springer, 117-136. http://www.usecasemaps.org/pub/SDL03-UCM-SDL.pdf.

[73] Charfi, L. (2001), Formal Modeling and Test Generation Automation with Use Case Maps and LOTOS. M.Sc. thesis, SITE, University of Ottawa, Canada, February 2001. http://www.usecasemaps.org/pub/lc_msc.pdf.

[74] Fernandez, J-C., Jard, C., Jéron, T., and Viho, C. (1996) Using On-the-fly Verification Techniques for the Generation of Test Suites. Computer Aided Verification (CAV'96), New Jersey, USA.

[75] Guan, R. (2002) From Requirements to Scenarios through Specifications: A translation Procedure from Use Case Maps to LOTOS. Master thesis, SITE, University of Ottawa, Canada. http://lotos.csi.uottawa.ca/ftp/pub/Lotos/Theses/rg_msc.doc.

[76] Bertolino, Antonia. "Software testing research: Achievements, challenges, dreams." 2007 Future of Software Engineering. IEEE Computer Society, 2007.

[77] Heumann, Jim. "Generating test cases from use cases." The rational edge 6.01 (2001).

[78] Ryser, J. and M. Glinz, 2000. SCENT: A method employing scenarios to systematically derive test cases for system test. Technical Report http://portal.acm.org/citation.cfm?id=901553.

[79] Nilawar, M. and S. Dascalu, 2003. A UML-based approach for testing web applications. M.Sc. Thesis, University of Nevada, Reno

[80] Hasling, B., Goetz, H., Beetz, K.: Model based testing of system requirements using UML use case models. In: 2008 1st International Conference on Software Testing, Verification, and Validation, pp. 367–376. IEEE (2008, April)

[81] Marrone, S., Flammini, F., Mazzocca, N., Nardone, R., Vittorini, V.: Towards model-driven V&V assessment of railway control systems. Int. J. Softw. Tools Technol. Transf. 16(6), 669–683 (2014)

[82] Heckel, R., Lohmann, M.: Towards model-driven testing. Electron. Notes Theor. Comput. Sci. 82(6), 33–43 (2003). ISBN 1571-0661

[83] Briand, L., Labiche, Y.: A UML-based approach to system testing. Softw. Syst. Model. 1(1), 10–42 (2002)

[84] Somé, S. S., Cheng, X.: An approach for supporting system-level test scenarios generation from textual use cases. In: Proceedings of the 2008 ACM Symposium on Applied computing, pp. 724–729. ACM (2008, March)

[85] C. Nebut, F. Fleurey, Y. Le Traon, et al., "Automatic test generation: a use case driven approach," IEEE Transactions on Software Engineering, vol. 32, pp. 140-55, 2006.

[86] Sebastian Siegl, Kai-Steffen Hielscher, and Reinhard German, "Model Based Requirements Analysis and Testing of Automotive Systems with Timed Usage Models," in 18th IEEE International Requirements Engineering Conference, Sydney, New South Wales Australia, 2010.

[87] Nogueira, S., Sampaio, A., Mota, A.: Test generation from state-based use case models. Formal Asp. Comput. 26(3), 441–490 (2014)

[88] Javed, A.Z., P.A. Strooper and G.N. Watson, 2007. Automated generation of test cases using model-driven architecture. Proceeding of the Second International Workshop on Automation of Software Test, May 20 - 26, Minneapolis, USA, 150-151.

[89] Sarmiento, E., Sampaio do Prado Leite, J. C., Almentero, E.: C&L: generating model-based test cases from natural language requirements descriptions. In: 2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET), pp. 32–38. IEEE (2014, August)

[90] Leite, J.C.S.P., Hadad, G., Doorn, J., Kaplan, G.: A scenario construction process. Requir. Eng. J. 5(1), 38–61 (2000)

[91] Tanvir Hussain and Robert Eschbach, "Automated Fault Tree Generation and Risk-Based Testing of Networked Automation Systems," in Proceedings of 15th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 10) Bilbao, Spain, 2010.

[92] Winkler, Stefan, and Jens von Pilgrim. "A survey of traceability in requirements engineering and model-driven development." Software & Systems Modeling 9.4 (2010): 529-565.

[93] Pinheiro, F.A.C.: Requirements traceability. In: Sampaio do Prado Leite, J.C., Doorn, J.H. (eds.) Perspectives on Software Requirements, pp. 93–113. Springer, Berlin (2003)

[94] Gotel, O.C.Z., Finkelstein, A.C.W.: An analysis of the requirements traceability problem. In: 1st IEEE International Requirements Engineering Conference (RE'94) Proceedings, pp. 94–101. IEEE Computer Society, New York (1994)

[95] Object Management Group: A Proposal for an MDA Foundation Model. Object Management Group, Needham, ormsc/05-04-01 ed. (2005)

[96] G. Spanoudakis, Zisman, A., Software Traceability: A Roadmap, Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing, 2005.

[97] BARMI, Zeinab Alizadeh, EBRAHIMI, Amir Hossein, et FELDT, Robert. Alignment of requirements specification and testing: A systematic mapping study. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. IEEE, 2011. p. 476-485.

[98] Duan, C., Cleland-Huang, J.: Visualization and analysis in automated trace retrieval. In: 1st International Workshop on Requirements Engineering Visualization (REV'06). IEEE Computer Society, New York (2006)

[99] F. Bouquet, Jaffuel, E., Legeard, B., Peureux, F., Utting, M., Requirements Traceability in Automated Test Generation - Application to Smart Card Software Validation, ICSE Int. Workshop on Advances in Model-Based Software Testing (A-MOST'05), ACM Press, St. Louis, USA, 2005

[100] Espinoza, A., Alarcon, P.P., Garbajosa, J.: Analyzing and systematizing current traceability schemas. In: Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop, pp. 21–32. IEEE Computer Society, New York (2006)

[101] Dahlstedt, Å.G., Persson, A.: Requirements interdependencies: state of the art and future challenges. In: Engineering and Managing Software Requirements, pp. 95–116. Springer, Berlin (2005). ISBN 978-3-540-25043-2

[102] F. Fraikin, Leonhardt, T., SeDiTeC — Testing Based on Sequence Diagrams, 17th IEEE International Conference on Automated Software Engineering, 2002, pp. 261 - 266.

[103] J. Wittevrongel, Maurer, F., SCENTOR: Scenario-Based Testing of E-Business Applications, Tenth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2001, pp. 41 - 46.

[104] L. C. Briand, Labiche, Y., A UML-Based Approach to System Testing, 4th International Conference on the Unified Modeling Language (UML), Toronto, Canada, 2001, pp. 194-208.

[105] F. Basanieri, Bertolino, A., Marchetti, E., The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects, Proceedings of the 5th International Conference on The Unified Modeling Language, Springer-Verlag, 2002, pp. 383-397.

[106] A. Hartman, Nagin, K., The AGEDIS tools for model-based testing, 2004 ACM SIGSOFT international symposium on Software testing and analysis, ACM Press, Boston, Massachusetts, USA, 2004, pp. 129-132.

[107] R. Marelly, D. Harel, and H. Kugler, "Multiple instances and symbolic variables in executable sequence charts," in 17th International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2002), USA, 2002, pp. 83-100.

[108] W. Grieskamp, Nachmanson, L., Tillmann, N., Veanes, M., Test Case Generation from AsmL Specifications - Tool Overview, 10th International Workshop on Abstract State Machines, Taormina, Italy, 2003.

[109] Naslavsky, Leila, Hadar Ziv, and Debra J. Richardson. "Towards traceability of model-based testing artifacts." Proceedings of the 3rd international workshop on Advances in model-based testing. ACM, 2007

[110] M. Felderer, P. Zech, F. Fiedler, et al., "A Tool based Methodology for System Testing of Service-oriented Systems," in Second International Conference on Advances in System Testing and Validation Lifecycle (VALID), Los Alamitos, CA, USA, 2010, pp. 108-13.

[111] F. Abbors, D. Truscan, and J. Lilius, "Tracing requirements in a model-based testing approach," in 2009 First International Conference on Advances in System Testing and Validation Lifecycle (VALID), Piscataway, NJ, USA, 2009, pp. 123-8.

[112] D. Arnold, J. P. Corriveau, and Shi Wei, "Modeling and validating requirements using executable contracts and scenarios," in 8th ACIS International Conference on Software Engineering Research, Management and Applications (SERA), CA, USA, 2010, pp. 311-20.

[113] A. Goel, B. Sengupta, and A. Roychoudhury, "Footprinter: Round-trip engineering via scenario and state-based models," in 31st International Conference on Software Engineering - Companion Volume - ICSE-Companion, Piscataway, NJ, USA, 2009, pp. 419-420.

[114] C. Pfaller, A. Fleischmann, J. Hartmann, et al., "On the integration of design and test: A model-based approach for embedded systems," in Proceedings of the 2006 international workshop on Automation of software test (AST) 2006, pp. 15-21.

[115] J. L. Boulanger and V. Q. Dao, "Requirements engineering in a model-based methodology for embedded automotive software," in IEEE International Conference on Research, Innovation and Vision for the Future in Computing 484 & Communication Technologies(RIVF), Ho Chi Minh City, Vietnam, 2008, pp. 263-268.

[116] H. Post, C. Sinz, F. Merz, et al., "Linking functional requirements and software verification," in 17th IEEE International Requirements Engineering Conference (RE), Piscataway, NJ, USA, 2009, pp. 295-302.

[117] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A subset of precise UML for model-based testing. In Proceedings of the 3rd International Workshop on Advances in Model-Based Testing, A-MOST ’07, pages 95–104, New York, NY, USA, 2007. ACM.

[118] J. Kelleher and M. Simonsson, "Utilizing use case classes for requirement and traceability modeling," in Proceedings of the 17th IASTED International Conference on Modelling and Simulation, Anaheim, CA, USA, 2006, pp. 617-25.

[119] A. Sabetta, D. C. Petriu, V. Grassi, et al., "Abstraction-raising transformation for generating analysis models," in Satellite Events at the MoDELS 2005 Conference. MoDELS 2005 International Workshops. Berlin, Germany, 2005, pp. 217-26.

[120] Tanvir Hussain and Robert Eschbach, "Automated Fault Tree Generation and Risk-Based Testing of Networked Automation Systems," in Proceedings of 15th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 10) Bilbao, Spain, 2010.

[121] S. Ibrahim, M. Munro, A. Deraman, et al., "A software traceability validation for change impact analysis of object-oriented software," in Proceedings of the International Conference on Software Engineering Research and Practice and Conference on Programming Languages and Compilers SERP'06, USA, 2006, pp. 453-9.

[122] Hubert Dubois, Marie-Agnès Peraldi-Frati, and Fadoi Lakhal, "A model for requirements traceability in an heterogeneous model-based design process. Application to automotive embedded

systems," in 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), Oxford, UK, 2010, pp. 233-242.

[123] M. W. Whalen, M. P. E. Heimdahl, A. Rajan, et al., "Coverage metrics for requirements-based testing," in Proceedings of the international symposium on Software testing and analysis (ISSTA) 2006, pp. 25-35.

[124] M. Conrad, I. Fey, and S. Sadeghipour, "Systematic Model-Based Testing of Embedded Automotive Software," Proceedings of the Workshop on Model Based Testing (MBT), Electronic Notes in Theoretical Computer Science, vol. 111, pp. 13-26, 2005.

[125] M. Riebisch and M. Hubner, "Traceability-driven model refinement for test case generation," in Proceedings. 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, CA, USA, 2005, pp. 113-20.

[126] Ulrich, A., Jell, S., Votintseva, A., & Kull, A. (2014, January). The ETSI Test Description Language TDL and its application. In Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on (pp. 601-608). IEEE.

[127] Philip Makedonski, Gusztav Adamis, Martti Käärik, Andreas Ulrich, Marc-Florian Wendland, Anthony Wiles. “Bringing TDL to users: A Hands-on Tutorial” User Conference on Advanced Automated Testing (UCAAT 2014), Munich.

[128] TTCN http://www.ttcn-3.org/index.php/tools

[129] Kesserwan, Nader, Rachida Dssouli, and Jamal Bentahar. "Modernization of Legacy Software Tests to Model-Driven Testing." International Conference on Emerging Technologies for Developing Countries. Springer, Cham, 2017.

[130] https://ucaat.etsi.org/2015/presentations/ESTERLINE_KESSERWAN.pdf

[131] Kesserwan, N., Dssouli, R., Bentahar, J., Stepien, B., & Labrèche, P. (2019). From use case maps to executable test procedures: a scenario-based approach. Software & Systems Modeling, 18(2), 1543-1570.

[132] Adolph, S., Cockburn, A., & Bramble, P. (2002). Patterns for effective use cases. Addison-Wesley Longman Publishing Co., Inc

[133] Kealey, J., Amyot, D.: Enhanced use case map traversal semantics. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745, pp. 133–149. Springer, Heidelberg (2007)

[134] http://xtext.com/.

[135] http://www.etsi.org/deliver/etsi_es/201800_201899/20187301/04.08.01_60/es_20187301v040801p.pdf.

[136] Boniol, F., Wiels, V.: The landing gear system case study. In: ABZ 2014: The Landing Gear Case Study, pp. 1–18. Springer (2014).

[137] Jouault, Frédéric, et al. "ATL: A model transformation tool." Science of computer programming 72.1-2 (2008): 31-39.

[138] D.J. Richardson, S.L. Aha, T.O. O’Malley, Specification-based test oracles for reactive systems, in: Proceedings of the 14th International Conference on Software Engineering, ICSE ’92, 1992, pp. 105–118.

[139] M. Didonet Del Fabro, Bézivin, J., Valduriez, P., Weaving Models with the Eclipse AMW plugin, Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany, 2006.

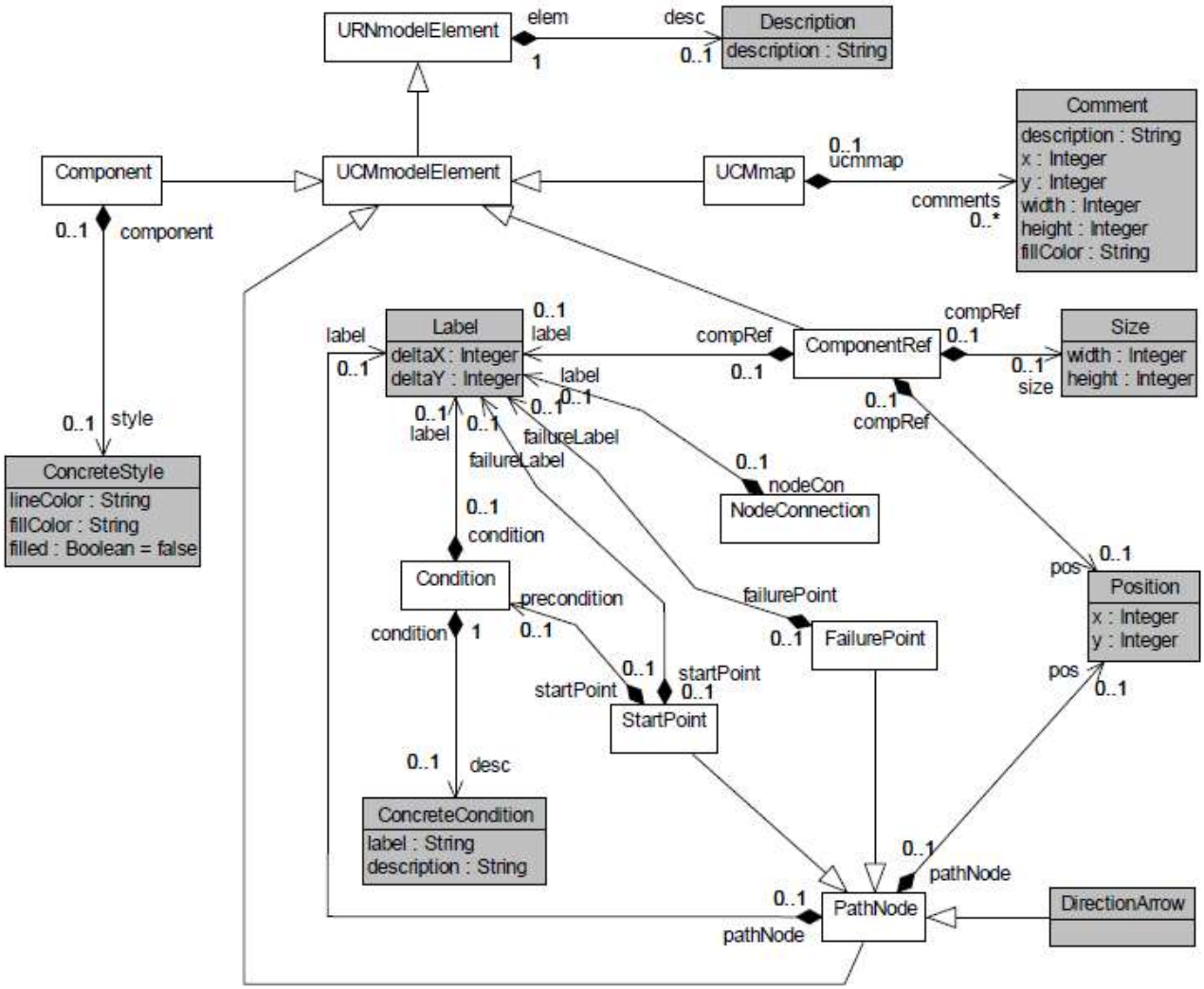
[140] Health, Social, and Economic Research, The economic impacts of inadequate infrastructure for software testing, National Institute of Standards and Technology, 2002.

[141] D. Jackson, M. Thomas, and L.I. Millett, Editors. Software for dependable systems: sufficient evidence? Committee on Certifiably Dependable Software Systems, National Research Council, National Academy of Sciences, 2007.

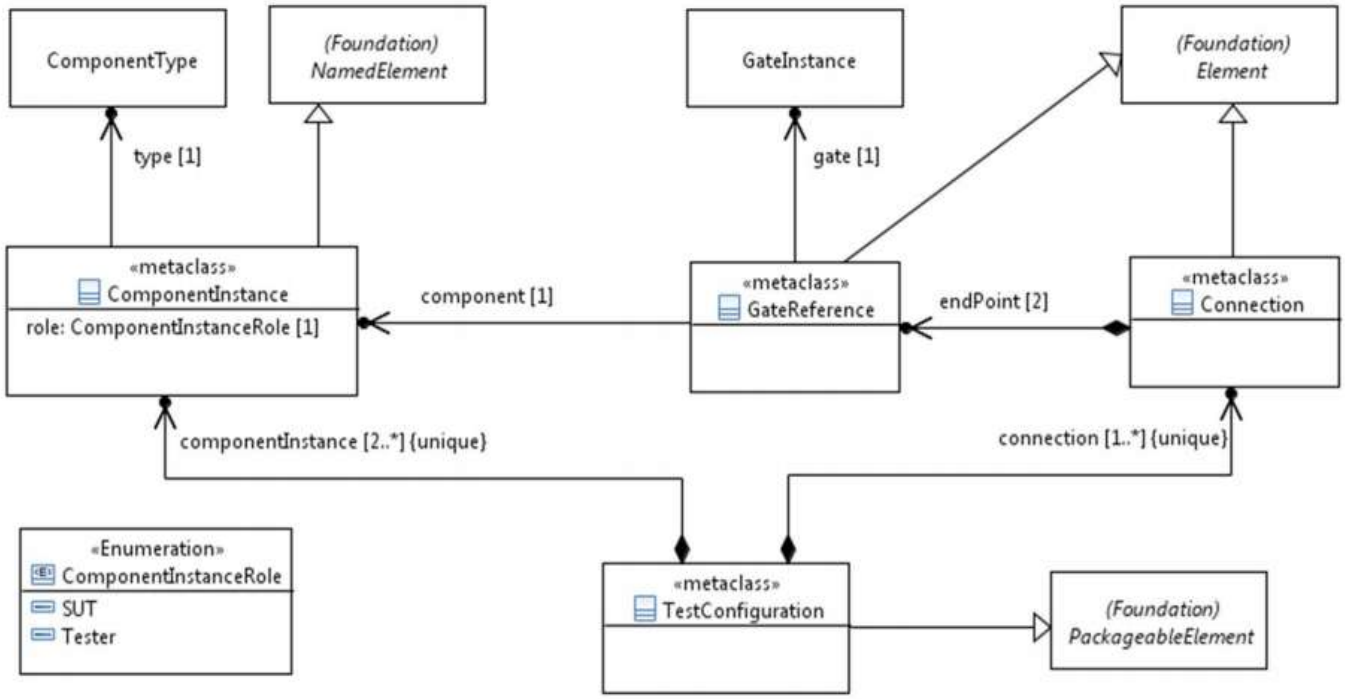
Appendices

Appendix A: UCM Metamodel

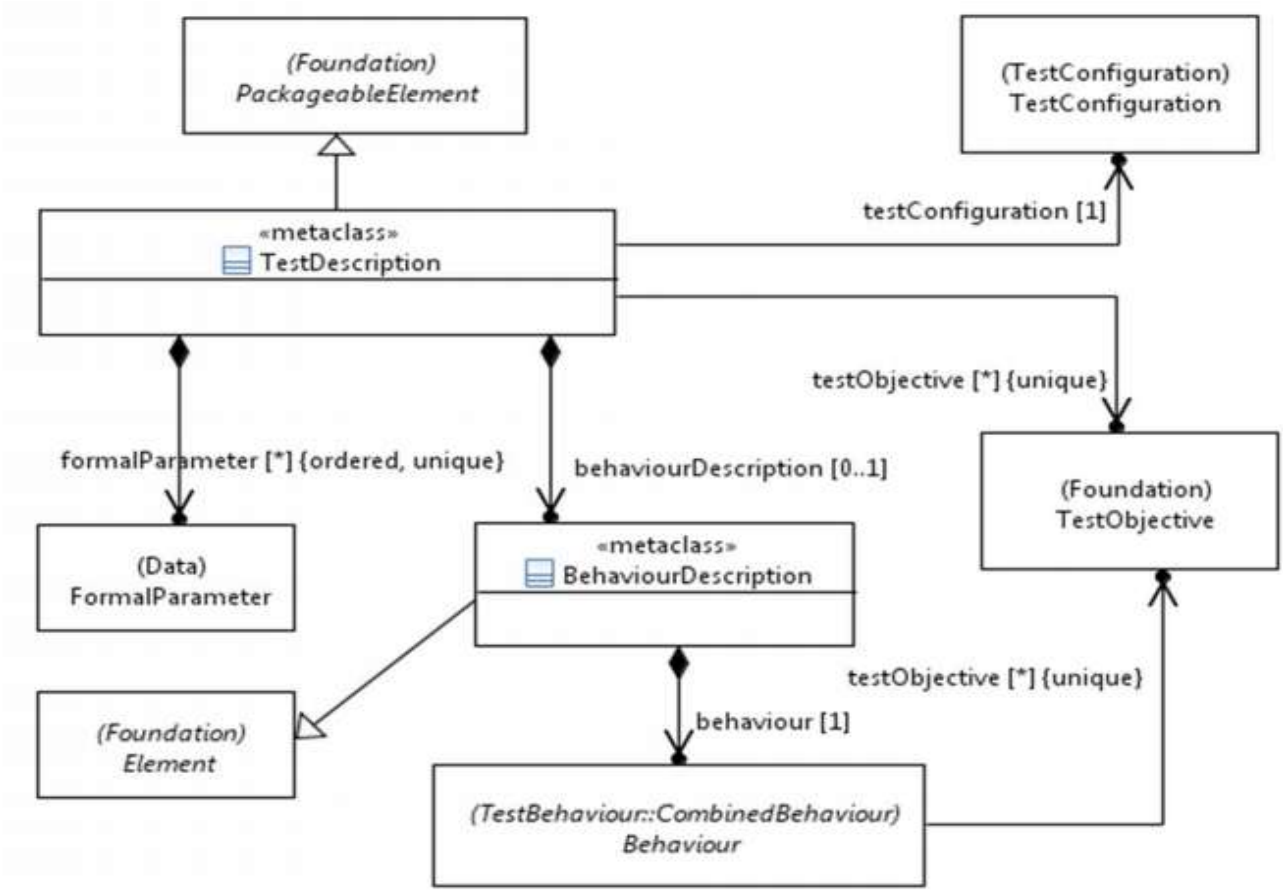
This Appendix presents the concrete metamodel of the UCM notation.



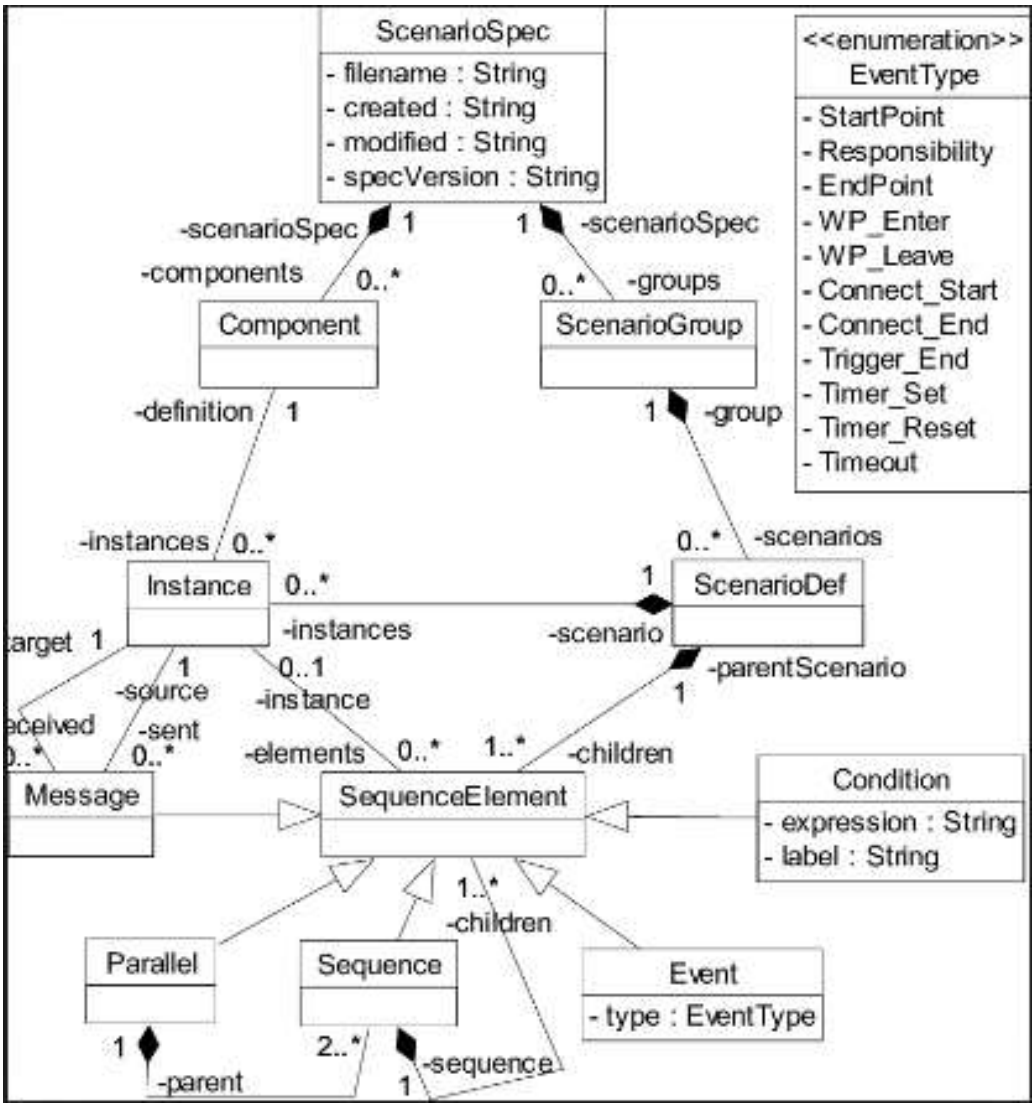
Appendix B: Test Configuration Metamodel



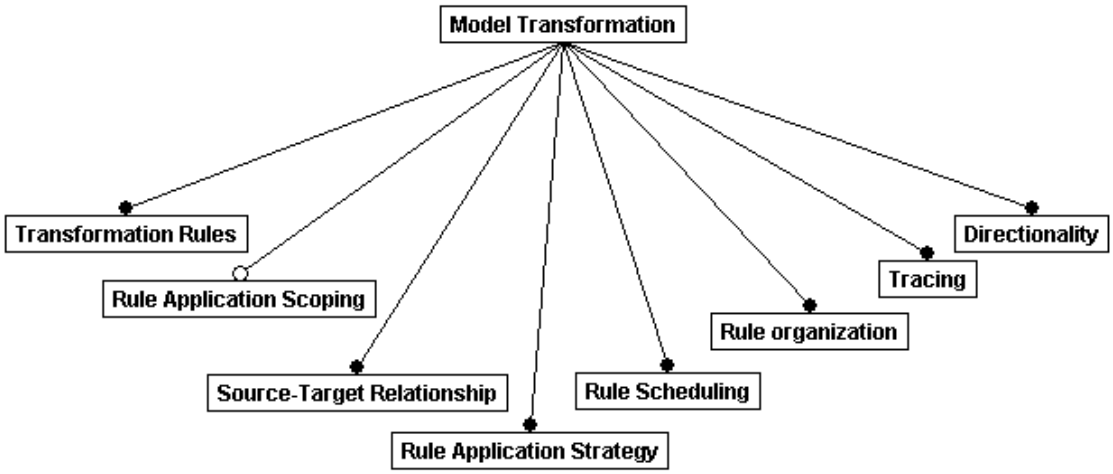
Appendix C: Test Description Metamodel



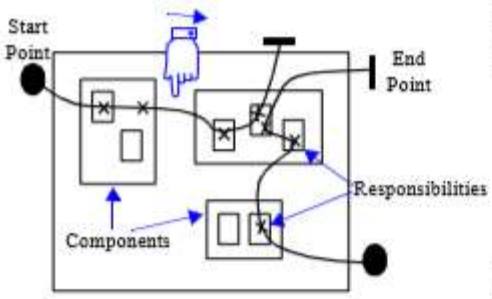
Appendix D: Scenario Definition Metamodel



Appendix E: Feature Model for Model Transformation



Appendix F: UCM Quick Reference Guide

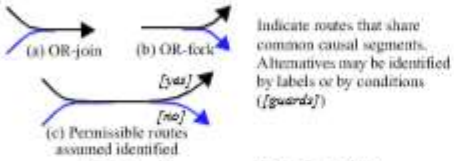


Imagine tracing a path through a system of objects to explain a causal sequence, leaving behind a visual signature. Use Case Maps capture such sequences. They are composed of:

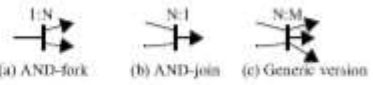
- **start points** (filled circles representing pre-conditions or triggering causes)
- causal chains of **responsibilities** (crosses, representing actions, tasks, or functions to be performed)
- and **end points** (bars representing post-conditions or resulting effects).

The responsibilities can be bound to **components**, which are the entities or objects composing the system.

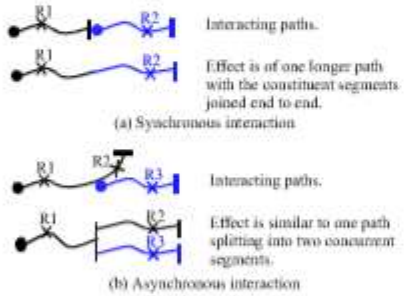
A1. Basic notation and interpretation



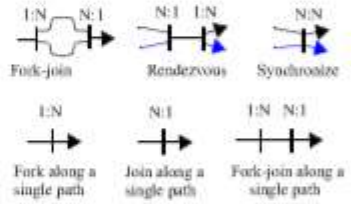
A2. Shared routes and **OR-forks/joins**.



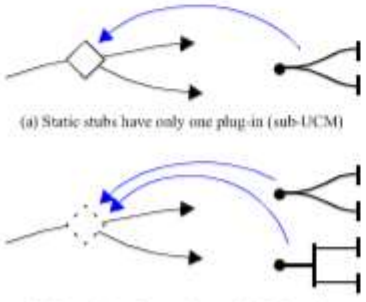
A4. Concurrent routes with **AND-forks/joins**.



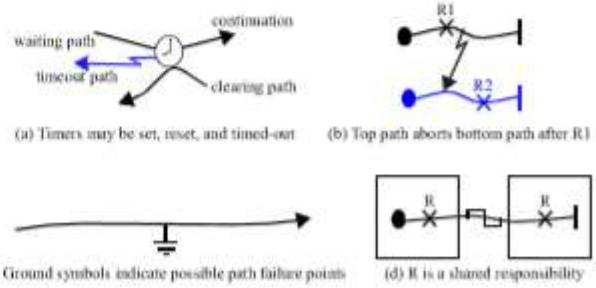
A3. Path interactions.



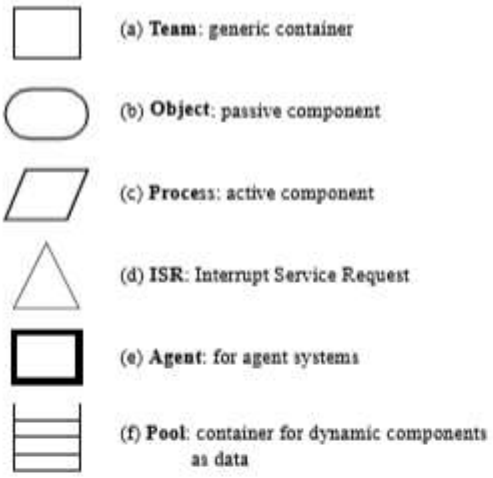
A5. Variations on AND-forks/joins.



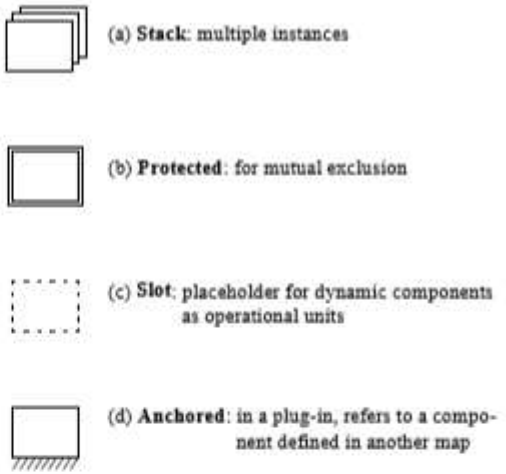
A6. **Stubs** and **plug-ins**.



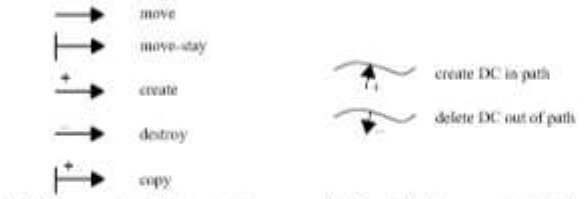
A7. **Timers, aborts, failures, and shared responsibilities.**

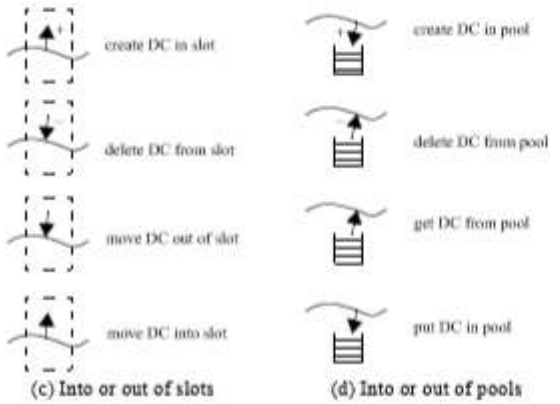


A8. Component types.

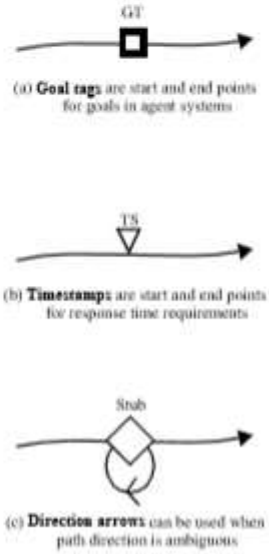


A9. Component attributes.





A10. Movement notation for **dynamic components** (DCs).



A11. Notation extensions